BASIC
Reference Manual

Order No. AA–L334A–TK
Including AD–L334A–T1

February 1984

This manual describes language elements, compiler commands, and compiler directives of VAX BASIC and PDP–11 BASIC–PLUS–2.

OPERATING SYSTEM AND VERSION:	VAX/VMS	V3
	RSX–11M–PLUS	V2
	RSX–11M	V4
	RSTS/E	V8
SOFTWARE VERSION:	VAX BASIC	V2
	PDP–11 BASIC–PLUS–2	V2

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage–paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

digital™	DECwriter	RSTS
	DIBOL	RSX
DEC	MASSBUS	UNIBUS
DECmate	PDP	VAX
DECsystem–10	P/OS	VMS
DECSYSTEM–20	Professional	VT
DECUS	Rainbow	Work Processor

Commercial Engineering Publications typeset this manual using DIGITAL's TMS–11 Text Management System.

Contents

PART II – Compiler Commands

PART III – Compiler Directives

PART IV – Statements

PART V – Functions

Tables

To the Reader

This manual is part of the BASIC documentation set. This set of manuals was designed to let you learn and use BASIC regardless of your prior experience with computers. The documentation set includes:

For the beginner:

- *Introduction to BASIC*
- *BASIC for Beginners*
- *More BASIC for Beginners*

For all systems:

- *BASIC User's Guide*
- *BASIC Reference Manual*
- *BASIC Pocket Reference Guide*

For specific systems:

- *BASIC on RSTS/E Systems*
- *BASIC on RSX–11M/M–PLUS Systems*
- *BASIC on VAX/VMS Systems*

For the system manager:

- *BASIC–PLUS–2 RSTS/E Installation Guide and Release Notes*
- *BASIC–PLUS–2 RSX–11M/M–PLUS Installation Guide and Release Notes*
- *VAX–11 BASIC Installation Guide and Release Notes*

For the beginner, *Introduction to BASIC* explains the fundamentals of the BASIC language and shows how to use BASIC to solve programming problems. *BASIC for Beginners* and *More BASIC for Beginners* lead the reader step-by-step through planning and writing several practical programs that teach BASIC programming techniques. In addition, the first chapter of the system-specific user's guide tells you how to log on to your computer system, create and execute programs, and do simple file operations such as printing, typing, and deleting files.

For programmers who are more familiar with BASIC, the *BASIC User's Guide* and the system-specific user's guides include a complete explanation of BASIC and how to use it on your system. If you need information on a particular feature or statement, the *BASIC Reference Manual* describes the format of each BASIC command or keyword individually.

The BASIC documentation set has several new features that let you find information quickly and easily. Each manual has its own index (with instructions on its use) and the *BASIC Reference Manual* has a master index to the entire documentation set. For quick reference the *BASIC Pocket Reference Guide* provides a brief explanation of all BASIC commands and functions. Similar information is also available at the computer terminal from the BASIC HELP facility.

The following pages describe the function of this particular manual. We welcome your comments and encourage you to use the Reader's Comments Form provided at the back of this book.

Document Objectives

This manual describes the language elements and syntax of Version 2 of *VAX–11 BASIC* and *BASIC–PLUS–2*. The term BASIC is used generically in this manual to refer to both *VAX–11 BASIC* and *BASIC–PLUS–2*. The term *VAX–11 BASIC* refers specifically to *VAX–11 BASIC* as implemented on VAX/VMS systems. *BASIC–PLUS–2* refers specifically to *BASIC–PLUS–2* as implemented on RSTS/E, RSX–11M, and RSX–11M–PLUS systems.

Note

For your convenience, examples, formats, or rules specific to *VAX–11 BASIC*, *BASIC–PLUS–2*, or *BASIC–PLUS–2* on *RSTS/E* or *RSX–11M/M–PLUS* are identified by a marginal symbol:

indicates *VAX–11 BASIC* only.

indicates *BASIC–PLUS–2* only.

indicates *BASIC–PLUS–2* on *RSTS/E* systems.

indicates *BASIC–PLUS–2* on *RSX–11M/M–PLUS* systems.

Intended Audience

This manual should be used by programmers familiar with computer concepts and the BASIC language. It is a reference manual to be used in conjunction with the BASIC user's guides.

Document Structure

This manual consists of six parts, two appendixes, and a master index to the BASIC documentation set. With the exception of Part I, BASIC language elements are arranged in alphabetical order within each part; each language element begins on a separate page. A sample format page is included on page xiv.

Part I Describes BASIC program elements and structure.

Part II Describes BASIC compiler commands.

Part III	Describes BASIC compiler directives.
Part IV	Describes BASIC statements.
Part V	Describes BASIC functions.
Part VI	Describes *BASIC–PLUS–2* debugger commands.
Appendix A	Lists reserved keywords.
Appendix B	Summarizes program and subprogram coding conventions.

This manual also includes three tabbed dividers for convenient reference:

- The first divider summarizes the conventions used in this manual.
- The second divider lists most BASIC keywords by function.
- The third divider precedes the Master Index and describes its use.

Sample Format Page

ENTRY NAME

1.0 ENTRY NAME

Function

Describes the entry's function or effect.

Format

A format shows the syntax of a language element. When you have a choice of formats, the formats are named for clarity. When a format is named *General*, it applies to both *VAX–11 BASIC* and *BASIC–PLUS–2*. Format components are explained in syntax and general rules. When a language element has more than one format, formats are referred to by name. Some formats are divided into two parts. The first part, in the top portion of the box, shows the general elements and order of the format.

The second part of the format, in the lower portion of the box, shows the components and order of the individual elements in the general format.

Syntax Rules

Syntax rules tell you how to order format elements to form clauses or statements. They also impose restrictions or relax restrictions implied by the format.

General Rules

General rules define the semantics of the entry and the entry's effect on program execution or compilation.

Examples

This section presents one or more sample program lines. All examples work for both *VAX–11 BASIC* and *BASIC–PLUS–2* unless otherwise noted.

Conventions

Formats present the correct syntax for writing BASIC source code. You must order syntax elements as shown in the format unless the syntax rules indicate otherwise.

Syntax formats consist of BASIC keywords, metalanguage mnemonics, and punctuation symbols. Metalanguage mnemonics are symbolic derivations of BASIC objects or structures. The tabbed divider that follows this section lists the most frequently used mnemonics and their meanings, as well as the most frequently used punctuation symbols.

Note

BASIC keywords are always capitalized in this manual and must be spelled exactly as shown. Mnemonics are in lowercase letters in formats and are italicized in the syntax and general rules.

Some metalanguage mnemonics are derived directly from BASIC keywords. For example:

- Map — From MAP
- Com — From COMMON
- Func — From FUNCTION
- Def — From DEF
- Sub — From SUB

Others are abbreviated forms of words. For example:

- Vbl — For variable
- Unsubs — For unsubscripted
- Subs — For subscripted
- Str — For string
- Const — For constant
- Exp — For expression
- Nam — For name
- Cond — For conditional
- Int — For integer
- File-spec — For file-specification
- Data-type — For data-type

Most mnemonics used in formats are combinations of mnemonics. For example:

- Const-nam — Is a constant name.
- Sub-nam — Is the name of a SUB subprogram.
- Unsubs-vbl — Is an unsubscripted variable.

(continued on next page)

- Int-exp — Is an integer expression.
- Cond-exp — Is a conditional expression.
- Str-unsubs-vbl — Is a string unsubscripted variable.

Mnemonics are combined in this way to indicate exactly what type of object or structure BASIC expects. Some BASIC statements, for example, allow you to specify any type of variable (string or numeric) in the format, while others allow only a numeric variable (integer or floating-point), a string variable, an integer variable, or a floating-point variable.

Thus, the uncombined form of the variable mnemonic (*vbl*) in a format means that you can use any type of variable (string or numeric). A combined variable mnemonic (such as *str-vbl*, *num-vbl*, or *int-vbl*) in a format means that you can specify only a particular type of variable.

Within formats, mnemonics are either simple or complex. Simple mnemonics identify a format element (such as an expression, a variable, or a name) that needs no further definition. For example:

```
EXTERNAL data-type CONSTANT const-nam,...
```

The mnemonics in this format need no further definition. The EXTERNAL keyword must be followed by a *data-type*, the CONSTANT keyword, and then a *const-nam*. The comma and ellipsis (...), as defined in the Punctuation Symbols Table, indicate that you can specify more than one *const-nam*. The *data-type* mnemonic is defined in the Mnemonics Table as a BASIC data-type keyword, and *const-nam* is defined as a constant name. Restrictions to the use of data-type keywords in the EXTERNAL statement are specified in the syntax rules.

Complex mnemonics identify a format element (such as a parameter passing mechanism or a statement clause) that has more than one component. Complex mnemonics are further defined in the lower portion of the format box by simple mnemonics. For example:

Format

```
Variables

    DECLARE data-type decl-item [, [ data-type ] decl-item ]...

DEF Functions

    DECLARE data-type FUNCTION { def-nam [ ( [ def-param ],... ) ] },...

Named Constants

    DECLARE data-type CONSTANT { const-nam = const },...
```

```
decl-item:    { unsubs-vbl-nam
              { array-nam ( int-const,... )

def-param:    [ data-type ]
```

When you look at the upper portion of this format, you can see that a data-type keyword must follow the DECLARE statement and that a *decl-item* must follow the data-type keyword. *Decl-item* is a complex mnemonic that is then further defined in the lower portion of the box. There you can see that a *decl-item* can be a simple variable name or an array name followed by parentheses and integer constants separated by parentheses. The portion of the upper format in brackets indicates that you can specify another data-type keyword and another array name or simple variable name. The comma and ellipsis (...), as defined on the tabbed divider in this section, indicate that you can continue adding data-type keywords and array names or simple variable names.

This type of format *unfolds* the syntax of BASIC language elements and indicates the type of element BASIC expects to receive.

Note

> In most cases, BASIC signals an error if the syntax element does not exactly match the indicated format. In other instances, particularly with numeric elements, BASIC converts the numeric element you specify to the type of numeric element it expects to receive. These instances are noted in the syntax rules.

Multiple occurrences of mnemonics in a format are numbered to prevent confusion. *Vbl3*, for example, is the third unique variable in a general format and is referred to as *vbl3* in the syntax and general rules.

The most frequently used punctuation symbols and metalanguage mnemonics are listed and described on the first tabbed divider in this manual. Less frequently used mnemonics and most complex mnemonics are defined as they occur in syntax formats.

Please use the Reader's Comments Form in the back of this book to report errors or to make suggestions for future documentation releases.

Conventions

Syntax Mnemonics

Mnemonic	Definition
exp	An expression
vbl	A variable
unsubs	Unsubscripted; used with the variable mnemonic to indicate a simple variable, as opposed to an array element
subs	Subscripted; used with the variable mnemonic to indicate an array element; the element's position in the array is specified by subscripts enclosed in parentheses and separated by commas
array	An array; syntax formats indicate whether you can specify bounds and dimensions, or just dimensions
const	A constant value
lit	A literal value, in quotation marks; a literal is always a constant, but a constant may be named, so constants are not always literals
num	A numeric value
real	A floating-point value
int	An integer value
str	A character string
cond	Conditional; used with the expression mnemonic to indicate that an expression can be either logical or relational
log	Logical; used with the expression mnemonic to indicate a logical expression
rel	Relational; used with the expression mnemonic to indicate a relational expression
lex	Lexical; used to indicate a component of a compiler directive
target	The target point of a branch statement; used to indicate that the target point can be either a program line number or a statement label
lin-num	A program line number
label	An alphanumeric statement label
item	Allowable BASIC objects, such as variables, data types, and parameters; allowable objects are defined in formats as they occur
nam	Name; indicates the declaration of a name or the name of a BASIC structure, such as a SUB subprogram
com	Specific to a COMMON
def	Specific to a DEF
func	Specific to a FUNCTION subprogram
map	Specific to a MAP
sub	Specific to a SUB subprogram
chnl	An I/O channel associated with a file
data-type	A data-type keyword
file-spec	A file-specification
file-nam	A file name

Punctuation Symbols

Symbols	Definition
[ ]	Brackets enclose an optional portion of a format. Brackets around vertically stacked entries indicate that you can select one of the enclosed elements. You must include all punctuation as it appears in the brackets.
{ }	Braces enclose a mandatory portion of a general format. Braces around vertically stacked entries indicate that you must choose one of the enclosed elements. Braces also group portions of a format as a unit. You must include all punctuation as it appears in the braces.
...	An ellipsis indicates that the immediately preceding language element can be repeated. An ellipsis following a format unit enclosed in brackets or braces means that you can repeat the entire unit. If repeated elements or format units must be separated by commas, the ellipsis is preceded by a comma (,...).

Definitions

In this manual, the following definitions apply:

BASIC	The term *BASIC* refers to Version 2 of both *VAX–11 BASIC* and *PDP–11 BASIC–PLUS–2*.
BASIC–PLUS–2	The term *BASIC–PLUS–2* refers specifically to Version 2 of *PDP–11 BASIC–PLUS–2* as implemented on *RSTS/E*, *RSX–11M*, and *RSX–11M–PLUS* systems.
Cannot	Cannot indicates than an operation cannot be performed and that an attempt to perform the operation causes BASIC to signal an error.
Cursor or cursor position	Cursor or cursor position refers to a terminal's print mechanism. It can be the flashing cursor on a video display terminal or the print head on a hard-copy terminal.
Must	Must indicates that an operation must be performed and that failure to perform the specified operation causes BASIC to signal an error.
Program module	A program module is a BASIC main program, a SUB subprogram, or a FUNCTION subprogram.
Subprogram	A separately compiled program module that must be linked or task-built with the main program.
Subroutine	A subroutine is a block of code accessed by a GOSUB or ON GOSUB statement. It is always in the same program module as the statement that accesses it.
VAX–11 BASIC	The term *VAX–11 BASIC* refers specifically to Version 2 of *VAX–11 BASIC* as implemented on *VAX/VMS* systems.

Functional List of BASIC Keywords

Arrays

DET
DIMENSION
MAT
MAT INPUT
MAT LINPUT
MAT PRINT
MAT READ
NUM
NUM2

Data Conversion

ASCII
CHANGE
CHR$
NUM$
NUM1$
STR$
VAL
VAL%

Data Definition

COMMON
DECLARE
DIMENSION
MAP
MAP DYNAMIC
MOVE
RECORD
REMAP

Data Formatting

FORMAT$
PRINT USING

Data Typing

COMMON
DECLARE
DEF
DIMENSION
EXTERNAL
FUNCTION
MAP
OPTION
SUB

Date and Time Conversion

DATE$
TIME
TIME$

Error Handling

ERL
ERN$
ERR
ERT$
ON ERROR GO BACK
ON ERROR GOTO
ON ERROR GOTO 0
RESUME

Function Definition

DEF
END DEF
END FUNCTION
EXIT DEF
EXIT FUNCTION
EXTERNAL
FUNCTION

I/O to Files

CLOSE
DELETE
FIND
FREE
GET
INPUT #
INPUT LINE #
KILL
LINPUT #
MAR
MARGIN
MOVE
NAME AS
OPEN
PRINT #
PUT #
RECOUNT
RESTORE #
SCRATCH
UNLOCK
UPDATE

I/O to Terminals

CCPOS
CTRLC
ECHO
INPUT
INPUT LINE
LINPUT
MAR
NOECHO
PRINT USING
RCTRLC
RCTRLO
RECOUNT
TAB

Numbers

ABS
ATN
COMP%
COS
DECIMAL
EXP
FIX
INT
INTEGER
LOG
LOG10
MAG
RANDOMIZE
REAL
RND
SGN
SIN
SQR
SWAP%
TAN

Program Control

END
EXIT LOOP
FOR
GOSUB
GOTO
IF
ITERATE
ON GOTO
RETURN
SELECT
SLEEP
STOP
UNLESS
UNTIL
WAIT
WHILE

Program Segmentation

CALL
CHAIN
END FUNCTION
END SUB
EXIT FUNCTION
EXIT SUB
EXTERNAL
FUNCTION
LOC
SUB

Strings

EDIT$
FORMAT$
INSTR
LEFT$
LEN
LSET
MID$
POS
RIGHT$
RSET
SEG$
SPACE$
STRING$
TRM$
XLATE

String Arithmetic

DIF$
PLACE$
PROD$
QUO$
SUM$

Value Assignment

DATA
LET
LSET
READ
RESTORE
RSET

PART I
Program Elements and Structure

1.0 Elements of a BASIC Program

A BASIC program is a series of program lines that contain instructions for the BASIC compiler. These instructions are in the form of BASIC statements. Program lines contain the BASIC keywords, operators, and operands that make up a BASIC program.

The first line of a BASIC program must begin with a line number. The program lines that follow may contain:

- Line numbers or labels
- Statements
- Optional compiler directives
- Optional comment fields
- Line terminator (carriage return)

1.1 Line Numbers

Every BASIC statement must be associated with a line number. Thus, the first element in a BASIC program must be a line number. A line number must be an integer between 1 and 32767, inclusive. A space or tab terminates the line number. Embedded spaces, tabs, and commas within line numbers are invalid.

A line number followed by a carriage return does not constitute a BASIC program line. A program line must contain a statement or a comment field. Comment fields are discussed in Section 2.1. A new line number or a carriage return terminates a BASIC program line.

A program line can contain any number of text lines; however, a text line cannot exceed 255 characters in *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSTS/E systems*, and 132 characters in *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems*.

The BASIC language uses line numbers to:

- Indicate the order of statement execution
- Provide control points for branching
- Help in debugging and updating programs
- Find the location of run-time errors
- Resume processing after an error has been handled

Therefore, each line number must be unique. BASIC ignores leading spaces, tabs, and zeros in line numbers.

1.2 Labels

A label is a 1– to 31–character name that immediately precedes a statement. It may immediately follow a line number. The label logically identifies a statement or block of statements. The label name must conform to the rules for naming variables, described in Section 6.1. The label name must be separated from the statement it labels with a colon (:). For example:

```
100     Yes_routine: PRINT "Your answer is YES."
```

The colon is not part of the label name. It tells BASIC that the label is being defined rather than referenced. Consequently, the colon is not allowed when you use a label to reference a statement. For example:

```
200     GOTO Yes_routine
```

The BASIC language uses labels to:

- Provide control points for branching
- Help in debugging programs
- Help in maintaining and updating programs

You can reference a label anywhere you can reference a line number, with three exceptions:

- You cannot compare the value returned by the ERL function (the line number associated with the program line where the last error occurred) with a label.
- You cannot use the RESUME statement to reference a label.
- You cannot reference a label in an IF–THEN–ELSE statement without using the keyword GOTO or GO TO. You can use the implied GOTO form only to reference a line number. For example:

```
100     IF A% = B%
           THEN 1000
           ELSE 1050

200     IF A$ = "YES"
           THEN GOTO Yes
           ELSE GOTO No
```

Because the first statement references a line number, the GOTO keyword is not required; the second statement references a label, so the GOTO keyword is required.

1.3 Statements

A BASIC statement consists of a statement keyword and optional operators and operands. For example:

```
400     LET A% = 534% + (SUM% - DIF%)
        PRINT A%
```

The first statement assigns a value to the integer variable A%. The PRINT statement causes BASIC to display the value of A% on your terminal.

A statement is either executable or nonexecutable:

- Executable statements perform operations (for example, PRINT, GOTO, and READ).
- Nonexecutable statements describe the characteristics and arrangement of data, specify usage information, and serve as comments in the source program (for example, DATA, DECLARE, and REM).

BASIC can accept and process one statement on a line of text, several statements on a line of text, multiple statements on multiple lines of text, and single statements continued over several lines of text. Each line of program text is associated with the last specified line number.

Multi-statement and continuation lines are discussed in Sections 1.3.2 and 1.3.3.

1.3.1 Keywords

A keyword is a reserved element of the BASIC language. Every statement except LET and empty statements must begin with a keyword. BASIC uses keywords to:

- Define data and user identifiers
- Perform operations
- Invoke built-in functions

Note

Keywords are reserved words and cannot be used as variable names or as names for MAP or COMMON areas.

Keywords cannot be used in any context other than as BASIC keywords. STRING$ = "YES", for example, is invalid because STRING$ is a reserved BASIC keyword. Appendix A in this manual contains a list of BASIC reserved keywords.

A BASIC keyword cannot have embedded spaces and cannot be split across lines of text. There must be a space, tab, or special character such as a comma between the keyword and any other variable or operator.

Some keywords use two words. In this case, their spacing requirements vary, as shown in Table 1.

Table 1: Keyword Space Requirements

Optional Space	Mandatory Space	No Space
GO SUB	BY DESC	FNEND
GO TO	BY REF	FNEXIT
ON ERROR	BY VALUE	FUNCTIONEND
	END DEF	FUNCTIONEXIT
	END FUNCTION	NOECHO
	END GROUP	NOMARGIN
	END IF	SUBEND
	END RECORD	SUBEXIT
	END SELECT	
	END SUB	
	EXIT DEF	
	EXIT FUNCTION	
	EXIT SUB	
	INPUT LINE	
	MAP DYNAMIC	
	MAT INPUT	
	MAT LINPUT	
	MAT PRINT	
	MAT READ	

1.3.2 Single-Statement Lines and Continued Statements

A single-statement line consists of one statement on one numbered line or one statement continued over two or more text lines. For example:

```
100     PRINT B * C / 12
```

This single-statement line has a line number, keyword (PRINT), operators (*, /), and operands (B, C, and 12).

You can have a single statement span several text lines by typing an ampersand (&) and a carriage return. For example:

```
100     OPEN "SAMPLE.DAT" AS FILE 2%, &(RET)
            SEQUENTIAL VARIABLE, &(RET)
            MAP ABC
```

The ampersand must come immediately before the carriage return in *VAX–11 BASIC*. *BASIC–PLUS–2* ignores spaces or tabs that follow the ampersand and precede the carriage return. For compatibility, DIGITAL recommends that you type the carriage return immediately after the ampersand.

The ampersand continuation character may be used but is not required for continued REM statements. The following example is valid:

```
100     REM This is a remark
            And this is also a remark
```

You can continue any BASIC statement, but you cannot continue a string literal or BASIC keyword. For example, BASIC returns the error message "Unterminated string literal" if you try to print the following:

```
100     PRINT "FEE-FIE-     &
               FOE-FUM"
```

This example is valid:

```
200     PRINT "FEE-";       &
              "FIE-";       &
              "FOE-";       &
              "FUM"
```

A more efficient way to continue string literals is to use the string concatenation operator:

```
100     PRINT "FEE-"        &
            + "FIE-"        &
            + "FOE-"        &
            + "FUM"
```

BASIC concatenates the four string literals at compile time and stores them as one string. When the PRINT statement executes, BASIC displays the one concatenated string literal rather than four separate string literals, thereby causing your program to execute faster and more efficiently.

Continued statements do not have line numbers, although the compiler counts and numbers them as sublines.

1.3.3 Multi-Statement Lines

Multi-statement lines contain several statements on one line of text or multiple statements on separate lines of text. All the statements on a multi-statement line are associated with a single line number.

Multiple statements on one line of text must be separated by backslashes (\). For example:

```
400     PRINT A \ PRINT V \ PRINT G
```

Because all statements are on the same program line, any reference to line number 400 refers to all three statements and execution begins with the first statement on the line. That is, BASIC cannot execute the second statement without executing the first statement.

A statement that unconditionally transfers control to another program line should always be the last statement on a multi-statement line. Otherwise, the statements that follow the statement transferring control will never execute. The following program line, for example, will execute, but it is not recommended:

```
200     PRINT A \ GOTO 410 \ PRINT B
```

BASIC prints the value of A and then branches to line 410. The statement PRINT B will never execute.

You can also write a multi-statement program line that associates all statements with a single line number by ending each statement with an ampersand (&) and a carriage return and preceding the next statement with a backslash. For example:

```
400     PRINT A     &
        \ PRINT V   &
        \ PRINT G
```

Because programs written in this format tend to be cluttered and hard to read, BASIC allows you to associate multiple statements with a line number by placing each statement on a separate line without using the ampersand or backslash. This format requires only a space or tab at the beginning of each new line of text. BASIC assumes that such an unnumbered line of text is either a new statement or an IF statement clause. For example:

```
400     PRINT A
        PRINT B
        PRINT "FINISHED"
```

In this example, each line of text begins with a BASIC statement and each statement is associated with line number 400.

BASIC also recognizes IF statement keywords on a new line of text and associates such keywords with the preceding IF statement. For example:

```
100     IF (A$ = "YES") OR (A$ = "Y")
        THEN PRINT "You typed YES"
        ELSE PRINT "You typed NO"
             STOP
        END IF
```

The BASIC compiler listing file numbers the lines associated with line number 100 as they occur. The *VAX–11 BASIC* listing file looks like this:


```
1 100         IF (A$ = "YES") OR (A$ = "Y")
2             THEN PRINT "You typed YES"
3             ELSE PRINT "You typed NO"
4                  STOP
5             END IF
```

The *BASIC–PLUS–2* listing file looks like this:


```
00001 100       IF (A$ = "YES") OR (A$ = "Y")
00002           THEN PRINT "You typed YES"
00003           ELSE PRINT "You typed NO"
00004                STOP
00005           END IF
```

Each statement has a number that indicates its position in the line. The BASIC compiler counts the statements in a multi-statement line to locate compile-time errors. You cannot use statement numbers as targets of branch statements. Targets of branch statements such as GOTO must be a line number or a label.

You can use any BASIC statement in a multi-statement line. However, a REM or DATA statement must be the last statement on a multi-statement line. This is because the compiler:

- Ignores all text following a REM keyword until it reaches a new line number.
- Treats all text following a DATA keyword as data until it reaches a new line number; thus, every DATA statement in your program has to have its own line number.

Because a leading space or tab not followed by a line number implies a new statement in a multi-statement line, compiler commands and immediate mode statements cannot be preceded by a space or tab. If you enter a compiler command or immediate mode statement, you cannot add more continuation lines to the last program line. If you attempt to do so, BASIC signals the error "unknown command input".

1.4 Compiler Directives

Compiler directives are instructions in a program that tell BASIC to perform certain operations as it compiles the program. With compiler directives, you can:

- Place program titles and subtitles in the header that appears on each page of the listing file
- Place a program version identification string in both the listing file and object module
- Start or stop the accumulation of listing information for selected parts of a program
- Start or stop the accumulation of cross-reference information for selected parts of a program
- Include BASIC code from another source file
- Conditionally compile parts of a program
- Terminate compilation
- Include CDD record definitions in a BASIC program (*VAX–11 BASIC* only)

All compiler directives:

- Must begin with a percent sign
- Can be preceded by an optional line number
- Must be the only text on the line (except for %IF–%THEN–%ELSE–%END–%IF)
- Must be preceded by a space, tab, or line number
- Cannot appear within a quoted string

See the *BASIC User's Guide* and Part III in this manual for more information on compiler directives.

1.5 Line Terminators

In the BASIC environment, a carriage return/line feed combination (RET) followed by an optional space or tab and a new line number ends a BASIC program line. An ampersand followed by a carriage return ends a line of text but not the program line. All statements between the first line number and the next line number are associated with the first line number.

1.6 Lexical Order

Lexical order refers to the order in which BASIC compiles statements in a program. In general terms, BASIC compiles program lines in sequential order from the lowest to the highest line number. Thus, statement A precedes statement B if the line number with which statement A is associated is lower than the line number with which statement B is associated. If both statements are associated with the same line number, statement A precedes statement B only if it physically precedes statement B or appears to the left of statement B. BASIC processes statements on a line of text from left to right and lines of text from top to bottom.

Some BASIC statements, such as comments and MAP declarations, are nonexecutable. If program control passes to a nonexecutable statement, BASIC executes the first statement that lexically follows the nonexecutable statement.

2.0 Program Documentation

Documentation clarifies and explains source program structure. You can provide such explanations with:

- Comment fields
- REM statements

2.1 Comment Fields

A comment field begins with an exclamation point (!) and ends with a carriage return. You supply text after the exclamation point to document your program. BASIC does not execute text in a comment field. For example:

```
100     ! FOR loop to initialize list Q
        FOR I = 1 TO 10
                Q(I) = 0 ! This is a comment
        NEXT I
        ! List now initialized
```

BASIC executes only the FOR loop. The comment fields, preceded by exclamation points, do not execute.

Comment fields help make your program more readable and allow you to format your program into readily visible logical blocks. They can also serve as target lines for GOTO and GOSUB statements:

```
10      !
        ! Square root program
        !
        INPUT 'Enter a number';A
        PRINT 'SQR of ';A;'is ';SQR(A)
        !
        ! More square roots?
        !
        INPUT 'Type "Y" to continue, a carriage return to quit';ANS$
        GOTO 10 IF ANS$ = 'Y'
        !
99      END
```

You can also use an exclamation point to terminate a comment field, but this practice is not recommended. Therefore, you should make sure that there are no exclamation points in the comment field itself; otherwise, BASIC treats the text remaining on the line as source code.

Note

Comment fields in DATA statements are invalid; the compiler treats the comments as additional data.

2.2 REM Statements

A REM statement begins with the REM keyword and ends when BASIC encounters a new line number. The text you supply between the REM keyword and the next line number documents your program. Like comment fields, REM statements do not affect program execution. BASIC ignores all characters between the keyword REM and the next line number. Therefore, the REM statement can be continued without the ampersand continuation character and should be the only statement on the line or the last of several statements in a multi-statement line:

```
10    REM This is an example
20    A=5
      B=10
      REM A equals 5
          B equals 10
30    PRINT A, B
```

The REM statement is nonexecutable. When you transfer control to the line number of a REM statement, BASIC executes the next executable statement that lexically follows the referenced line. For example:

```
10    REM ** Square root program
20    INPUT 'Enter a number';A
      PRINT 'SQR of ';A;'is ';SQR(A)
      INPUT 'Type "Y" to continue, a carriage return to quit';ANS$
      GOTO 10 IF ANS$ = 'Y'
40    END
```

When the conditional GOTO statement in line 20 transfers program control to line 10, BASIC ignores the REM comment on line 10 and continues program execution at line 20.

Note

Because BASIC treats all text between the REM statement and the next line number as commentary, REM should be used very carefully in programs that follow the implied continuation rules. Program statements intended for execution will not execute when they are inside a REM statement. DIGITAL recommends the use of comment fields (!) for program documentation in programs formatted with implied continuation lines.

2.3 Empty Statements

Empty statements consist of a line number and an exclamation mark followed by optional text, a line terminator and a new line number. For example:

```
100     !
        ! FOR loop to initialize list Q
        !
200     FOR I = 1 TO 10
                Q(I) = 0 ! This is a comment
        NEXT I
300     !
        ! List is now initialized
```

Lines 100 and 300 are empty statements.

3.0 BASIC Character Set

BASIC uses the full ASCII character set. This includes:

- The letters A through Z, both upper- and lowercase
- The digits 0 through 9
- Special characters

Appendix C in *BASIC on VAX/VMS Systems, BASIC on RSX–11M/M–PLUS Systems,* and *BASIC on RSTS/E Systems* contains the full ASCII character set and character values.

The compiler:

- Does not distinguish between upper- and lowercase letters except in string literals or within a DATA statement
- Does not process nonprinting characters unless they are part of a string literal
- Does not process characters in REM statements or comment fields

In string literals, BASIC processes:

- Lowercase letters as lowercase
- Nonprinting characters

The ASCII character NUL (ASCII code 0) and line terminators cannot appear in a string literal. Use the CHR$ function or explicit literal notation to use this character and terminators.

You can use nonprinting characters in your program, for example, in string constants, but to do so you must use: 1) a predefined constant such as ESC and DEL, 2) the CHR$ function to specify an ASCII value, or 3) explicit literal notation for character constants. See Section 5.4 in this manual for more information on explicit literal notation. See the *BASIC User's Guide* for more information on predefined constants and the CHR$ function.

4.0 BASIC Data Types

All data in a BASIC program has a specific data type that determines how many bits of storage should be considered as a unit and how the unit is to be interpreted and manipulated.

VAX–11 BASIC recognizes five primary data types: integer, floating-point, character string, packed decimal, and RFA. These types correspond to the BASIC generic data-type keywords:

- INTEGER
- REAL
- STRING
- DECIMAL
- RFA

BASIC–PLUS–2 recognizes four primary data types: integer, floating-point, character string, and RFA. These types correspond to the BASIC generic data-type keywords:

- INTEGER
- REAL
- STRING
- RFA

Integer data are stored as binary values in a byte, a word, or a longword. These values correspond to the BASIC data-type keywords:

- BYTE
- WORD
- LONG

Floating-point values are stored using a signed exponent and a binary fraction. *VAX–11 BASIC* allows four floating-point formats: single, double, gfloat, and hfloat. These formats correspond to the BASIC data-type keywords:

- SINGLE
- DOUBLE
- GFLOAT
- HFLOAT

BASIC–PLUS–2 allows only single and double floating-point formats. These formats correspond to the BASIC data-type keywords:

- SINGLE
- DOUBLE

VAX–11 BASIC packed decimal data is stored in a string of bytes. Refer to Appendix C in *BASIC on VAX/VMS Systems* for more information on the storage of packed decimal data.

Character data are strings of bytes containing ASCII codes as binary data. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. *VAX–11 BASIC* allows up to 65535 characters for a STRING data element. *BASIC–PLUS–2* allows up to 32767 characters.

In addition to these data types, BASIC also recognizes a special RFA data type to provide information about a Record File Address (RFA). A Record File Address consists of a block number within a file and an offset into that block. An RFA uniquely identifies a record in a file. You can access RMS files of any organization by Record File Address (RFA). This means that you specify the disk address of a record, and RMS retrieves the record at that address. Accessing records by RFA is more efficient and faster than other forms of random record access.

The RFA data type is unique and can be used only for:

- RFA operations (with the GETRFA function and GET and FIND statements)
- Assignments to other variables of the RFA data type
- Comparisons with other variables of the RFA data type using the equal to (=) or not equal to (<>) relational operators
- Formal and actual parameters
- DEF and function results

You cannot use variables or constants of the RFA data type for any arithmetic operations. You cannot declare a constant of the RFA data type.

The RFA data type requires six bytes of information: four bytes for the address of a disk block, and two bytes for the offset into the disk block. See Chapter 9 in the *BASIC User's Guide* for more information on Record File Addresses and the RFA data type.

Table 2 lists BASIC data-type keywords and summarizes BASIC data types.

Table 2: BASIC Data Types

Data Type Keyword*	Size	Range**	Precision (decimal digits)
INTEGER – specifies integer data			
BYTE	8 bits	–128 to +127	NA
WORD	16 bits	–32768 to +32767	NA
LONG	32 bits	–2147483648 to +2147483647	NA
REAL – specifies floating-point data			
SINGLE	32 bits	$.29 * 10^{-38}$ to $1.7 * 10^{38}$	6
DOUBLE	64 bits	$.29 * 10^{-38}$ to $1.7 * 10^{38}$	16
GFLOAT	*64 bits*	$.56 * 10^{-308}$ to $.9 * 10^{308}$	*15*
HFLOAT	*128 bits*	$.84 * 10^{-4932}$ to $.59 * 10^{4932}$	*33*
DECIMAL(d,s)	*0 to 16 bytes*	$1 * 10^{-31}$ to $1 * 10^{31}$	*31*
STRING	One character per byte	NA	NA
RFA	6 bytes	NA	NA

* *VAX–11 BASIC only data types are italicized.*

** Approximate for REAL and DECIMAL data types.

For the *VAX–11 BASIC* only DECIMAL data type, you can specify the total number of digits (d) in the data type and the number of digits to the right of the decimal point (s). For instance, DECIMAL(10,3) specifies decimal data with a total of 10 digits, 3 of which are to the right of the decimal point.

In Table 2, REAL and INTEGER are generic data-type keywords that specify floating-point and integer storage, respectively. If you use the REAL or INTEGER keywords to type data, the actual data type (SINGLE, DOUBLE, *GFLOAT* or *HFLOAT* in *VAX–11 BASIC*, BYTE, WORD, or LONG) depends on the current default. That is, if you do not explicitly type REAL and INTEGER data as SINGLE, DOUBLE, BYTE, WORD, and so on, BASIC uses the current defaults for REAL and INTEGER.

You can specify data-type defaults in the BASIC environment with the SET and COMPILE commands or in a program module with the OPTION statement. On *VAX/VMS systems*, you can also specify data-type defaults from DCL level with the DCL BASIC command. You can also specify whether program values are to be typed implicitly or explicitly. The following sections discuss data-type defaults and implicit and explicit data typing.

4.1 Implicit Data Typing

You implicitly assign a data format to program values by adding a suffix to the variable name or constant value or by specifying no suffix with the variable name or constant value:

- A dollar sign suffix ($) specifies STRING storage.
- A percent sign suffix (%) specifies INTEGER storage.
- No suffix character specifies storage of the default type, which may be INTEGER, REAL, or DECIMAL (*VAX–11 BASIC* only).

Suffixes on variable names and program constants specify string, integer, or floating-point storage of the default size. No suffix character implies that the value is of the default type (integer, floating-point, or packed decimal in *VAX–11 BASIC*). With implicit data typing, the range and precision for integer, floating-point, and packed decimal values (*VAX–11 BASIC* only) is determined by the current default data type. The default data type is determined by the system default (REAL) or the data type set for the BASIC environment with the SET or COMPILE commands. *VAX–11 BASIC* qualifiers are described in Table 16. *BASIC–PLUS–2* qualifiers are described in Table 17.

Note that if you compile your program with the /TYPE: EXPLICIT qualifier, you cannot type program values implicitly. All program values must be explicitly assigned a data type in your program or BASIC signals an error.

Good programming practice dictates that you do not mix implicit and explicit data typing in expressions or in program units and that you do not rely extensively on implicit data typing. Explicit data typing makes programs easier to understand and maintain because the data type of all program values is explicitly spelled out in the program and is not as dependent upon compilation defaults that may change.

4.2 Explicit Data Typing

Explicit data typing means that you use a declarative statement to specify the type, range and precision of your program values. Declarative statements associate attributes such as data type and value with user identifiers. For example:

```
100     DECLARE STRING CONSTANT ZIP_CODE = 03060
        DECLARE STRING EMP_NAME, DOUBLE WITH_TAX, SINGLE INT_RATE
```

The first DECLARE statement associates the constant value 03060 and the STRING data type with a constant named ZIP_CODE. The second DECLARE statement associates the STRING data type with EMP_NAME, the DOUBLE data type with WITH_TAX, and the SINGLE data type with INT_RATE. No constant values are associated with user identifiers in the second DECLARE statement because they are variable names.

With explicit data typing, each program variable within a program can have a different range and precision. This gives you more control over your program. Because you can explicitly assign data types to variables, constants, arrays, parameters, and functions, all integer data, for instance, does not have to take the compilation defaults. Likewise, all floating-point data does not have to take the compilation default because you can declare floating-point values as SINGLE or DOUBLE in *BASIC–PLUS–2* and as SINGLE, DOUBLE, GFLOAT, or HFLOAT in *VAX–11 BASIC*. See the *BASIC User's Guide* and the sections on these statements in this manual for more information on explicitly typing data.

Using the REAL and INTEGER keywords to explicitly type program values allows you to write programs that are transportable across systems, since these data-type keywords specify that all floating-point and integer data take the current default for REAL and INTEGER. The data type INTEGER, for example, specifies only that the constant or variable is an integer. The actual subtype (BYTE, WORD, or LONG) depends on the default set with the COMPILE or SET command, the *VAX–11 BASIC* DCL BASIC command, or the OPTION statement.

You can also specify a particular data type size for values declared INTEGER or REAL with compilation qualifiers. The qualifier /DOUBLE, for instance, specifies that all data typed REAL is to be treated as double-precision data.

The /TYPE: EXPLICIT qualifier or OPTION TYPE = EXPLICIT statement allows you to specify that all program data must be explicitly typed. Compiling a program with /TYPE: EXPLICIT or specifying OPTION TYPE = EXPLICIT means that any program value not explicitly declared causes BASIC to signal an error.

For new applications, DIGITAL recommends using BASIC's explicit data typing features. See Chapter 5 of the *BASIC User's Guide* for more information.

5.0. Constants

A constant is a numeric or character literal that does not change during program execution. A constant can also be named and associated with a data type. BASIC allows the following types of constants:

- Numeric

  Floating-point

  Integer

  Packed decimal (*VAX–11 BASIC* only)

- String (ASCII characters enclosed in quotation marks)

A constant of any of the above data types can be named with the DECLARE CONSTANT statement. You can then refer to the constant by name in your program. Refer to Section 5.3 for information on naming constants.

You can also use a special explicit literal notation to specify the value and data type of a numeric literal. Explicit literal notation is discussed in Section 5.4.

If you do not specify a data type for a numeric constant with the DECLARE CONSTANT statement or with explicit literal notation, the type and size of the constant is determined by the default REAL, INTEGER, or (*VAX–11 BASIC* only) DECIMAL set:

- At installation (*BASIC–PLUS–2* only)
- With the DCL BASIC command (*VAX–11 BASIC* only)
- With the SET command
- With the COMPILE command
- With the OPTION statement

BASIC also supplies predefined constants for ease in representing some ASCII characters and mathematical values.

The following sections discuss numeric and string constants, named constants, explicit literal notation, and predefined constants.

5.1 Numeric Constants

A numeric constant is a literal or named constant whose value never changes. In *VAX–11 BASIC*, a numeric constant can be a floating-point number, an integer, or a packed decimal number. In *BASIC–PLUS–2*, a numeric constant can be either a floating-point number or an integer. The type and size of numeric constants are determined by the current default values, the data-type qualifiers specified with the COMPILE command, the defaults set by the SET command, the data type specified in a DECLARE CONSTANT or OPTION statement, or by explicit literal notation.

If you use a declarative statement to declare data type and name a numeric constant, the constant is of the type and size specified in the statement. For example:

```
100     DECLARE BYTE CONSTANT AGE = 12
```

This example associates the numeric literal 12 and the BYTE data type with the user identifier AGE. To specify a data type for unnamed numeric constants, you must use the explicit literal notation format described in Section 5.4.

5.1.1 Floating-Point Constants

A floating-point constant is a literal or named constant with one or more decimal digits, either positive or negative, an optional decimal point and an optional exponent (E notation). If the default data type is INTEGER, a decimal point or an E is required or BASIC treats the literal as an INTEGER. In *VAX–11 BASIC*, if the default data type is DECIMAL, an E is required or *VAX–11 BASIC* treats the literal as a packed decimal value. The following, for example, are REAL literals:

Default type REAL:

```
    –8.738
239.21E–6
      .79
      299
```

Default type INTEGER:

```
    -8.738
239.21E-6
      .79
     299E
```

Default type DECIMAL (*VAX-11 BASIC* only):

```
    -8.738E
239.21E-6
      .79E
     299E
```

Very large and very small numbers can be represented in E (exponential) notation. This form of mathematical shorthand uses the format:

± number E ± n

where:

+ or – Is the number's sign. The plus sign is optional, but negative numbers require a minus sign.

number Is the number carried to a maximum of:

- 6 decimal places for SINGLE precision
- 16 decimal places for DOUBLE precision
- 15 decimal places for GFLOAT precision (*VAX-11 BASIC* only)
- 33 decimal places for HFLOAT precision (*VAX-11 BASIC* only)

E Represents the words "times 10 to the power of."

+ or – Is the exponent's sign. The plus sign is optional, but the minus sign is mandatory for negative exponents.

n Is an integer constant (the power of 10). If an exponent sign is specified, n can be zero, but not blank. If an exponent sign is not specified, n can be blank.

Table 3 compares numbers in standard and E notation.

Table 3: Numbers in E notation

Standard Notation	E Notation
.0000001	.1E-06
1,000,000	.1E+07
-10,000,000	-.1E+08
100,000,000	.1E+09
1,000,000,000,000	.1E+13

The range and precision of floating-point constants are determined by the current default data types or the explicit data type used in the DECLARE CONSTANT statement. There are, though, limits to the range allowed for numeric data types. Table 2 lists BASIC data types and ranges. BASIC signals the fatal error "floating point error or overflow" when your program specifies a constant value outside of the allowable range for a floating-point data type.

5.1.2 Integer Constants

An integer constant is a literal or named constant, either positive or negative, with no fractional digits and an optional trailing percent sign (%). The percent sign is required for integer literals if the default type is not INTEGER. For example:

Default type INTEGER:

```
81257
-3477
   79
```

Default type REAL or (*VAX–11 BASIC* only) DECIMAL:

```
81257%
-3477%
   79%
```

The range of allowable values for integer constants is determined by either the current default data type or the explicit data type used in the DECLARE CONSTANT statement. Table 2 lists BASIC data types and ranges. BASIC signals an error for a number outside the applicable range.

BASIC treats numeric literals as floating-point numbers unless:

- The default data type is INTEGER
- The literal has a % suffix

Thus, BASIC must convert numeric literals when assigning them to integer variables. This means that your program runs somewhat slower than it would if integer values were explicitly declared. You can prevent this conversion step by using percent signs for integer constants, numeric literal notation, or named integer constants.

Note

You cannot use percent signs in integer constants that appear in DATA statements. An attempt to do so causes BASIC to signal "Data format error" (ERR=50).

5.1.3 Packed Decimal Constants (VAX–11 BASIC Only)

A packed decimal constant is a number, either positive or negative, that has a specified number of digits and a specified decimal point position (scale). You specify the number of digits (d) and the position of the decimal point (s) when you declare the constant as a DECIMAL. If the constant is not declared, the number of digits and the position of the decimal are determined by the representation of the constant. For example, when the default data type is DECIMAL, 1.234 is a DECIMAL(4,3) constant, regardless of the default decimal size. Likewise, using explicit literal notation, "1.234"P is a

DECIMAL(4,3) constant, regardless of the default data type and default DECIMAL size. Explicit literal notation is described in Section 5.4. See the *BASIC User's Guide* for more information on packed decimal numbers.

5.2 String Constants

String constants are either string literals or named constants. A string literal is a series of characters enclosed in string delimiters. Valid string delimiters are:

- Double quotation marks ("text")
- Single quotation marks ('text')

You can embed double quotation marks within single quotation marks ('this is a "text" string') and vice versa ("this is a 'text' string"). Note, however, that BASIC does not accept incorrectly paired quotation marks and that only the outer quotation marks must be paired. The following character strings, for example, are valid:

"The record number does not exist."

"I'm here!"

"The terminating 'condition' is equal to A$."

"REPORT 543"

The following strings are not valid:

"Quotation marks do not match'

"No closing quotation mark

Characters in string constants can be letters, numbers, spaces, tabs, or any ASCII character except a line terminator or NUL (ASCII code 0). If you need a string constant that contains a NUL, you should use the NUL predefined constant in a compile-time constant expression or explicit literal notation. See Section 5.4 in this manual for information on explicit literal notation and the *BASIC User's Guide* for more information on the NUL predefined constant.

BASIC determines the value of the string constant by scanning all its characters. For example, because of the number of spaces between the delimiters and the characters, these two string constants are not the same:

" END-OF-FILE REACHED "

"END-OF-FILE REACHED"

BASIC stores every character between delimiters exactly as you type it into the source program, including:

- Lowercase letters (a–z)
- Leading, trailing, and embedded spaces
- Tabs
- Special characters

BASIC does not print the delimiting quotation marks when executing the program. That is, the value of the string constant does not include the delimiting quotation marks. For example:

```
100     PRINT "END-OF-FILE REACHED"
         !
         !
         !
200     END
RUNNH
END-OF-FILE REACHED
```

BASIC prints double or single quotation marks when they are enclosed in a second paired set:

```
100     PRINT 'FAILURE CONDITION: "RECORD LENGTH"'
         !
         !
         !
200     END
RUNNH
FAILURE CONDITION: "RECORD LENGTH"
```

5.3 Named Constants

BASIC allows you to name constants. You can assign a mnemonic name to a constant that is internal to your program and refer to the constant by name throughout the program. You can also name a constant that is external to your program and refer to it by name throughout your program. This naming feature is useful for the following reasons:

- If a commonly-used constant must be changed, you need to make only one change in your program.
- A logically named constant makes your program easier to understand.

You can use named constants anywhere you can use a constant, for example, to specify the number of elements in an array.

You cannot change the value of an explicitly named constant during program execution. To change the value of a constant, you must change the program statement that names the constant and declares its value and then recompile the program.

5.3.1 Naming Constants Within a Program Unit

You name constants within a program unit with the DECLARE statement. For example:

```
100     DECLARE DOUBLE CONSTANT Preferred_rate = .147
        DECLARE SINGLE CONSTANT Normal_rate = .162
        DECLARE DOUBLE CONSTANT Risky_rate = .175
            !
            !
            !
500     New_bal = Old_bal * (1 + Preferred_rate)^Years_payment
```

When interest rates change, only three lines have to be changed rather than every line that contains an interest rate constant.

Constant names must conform to the rules for naming internal, explicitly declared variables listed in Section 6.1. No constant name can have embedded spaces.

The value associated with a named constant can be a compile-time expression as well as a literal value. For example:

```
100     DECLARE STRING CONSTANT Congrats =      &
         "+--------------------+" + LF + CR +   &
         "! Congratulations!   !" + CR + CR +   &
         "+--------------------+"
        !
        !
        !
500     PRINT Congrats
        !
        !
        !
1000    PRINT Congrats
```

Named constants can save you programming time (since you don't have to retype the congratulations box every time you want to display it) and execution time (since the named constant is known at compile time).

Allowable operators in DECLARE CONSTANT expressions include all valid arithmetic, relational, and logical operators except exponentiation. You cannot use built-in functions in DECLARE CONSTANT expressions.

BASIC–PLUS–2 allows you to name floating-point, integer, and string constants, but floating-point constants cannot be named as expressions. Only STRING and INTEGER constants can be named as expressions in DECLARE CONSTANT statements. *VAX–11 BASIC* allows constants of all data types to be named as expressions. For example:

```
100     DECLARE DOUBLE CONSTANT    &
                MIN_VALUE = 0,     &
                MAX_VALUE = PI/2
```

This example is valid only in *VAX–11 BASIC*.

Note that you can specify only one data type in a DECLARE CONSTANT statement. To declare a constant of a different data type, you must use a second DECLARE CONSTANT statement.

5.3.2 Naming Constants External to a Program Unit

To declare constants outside the program unit, use the EXTERNAL statement. For example:

```
200     EXTERNAL LONG CONSTANT SS$_NORMAL
        EXTERNAL WORD CONSTANT IS.SUC
```

The first line declares the *VAX/VMS* status code SS$_NORMAL to be an external LONG constant. The second line declares IS.SUC, a success code, to be an external WORD constant. Note that *VAX–11 BASIC* allows external BYTE, WORD, LONG, and SINGLE constants, while *BASIC–PLUS–2* allows only external WORD constants. The linker or task builder supplies the values for the constants specified in EXTERNAL statements.

External constant names cannot exceed six characters in *BASIC–PLUS–2* and 31 characters in *VAX–11 BASIC* and must conform to the rules for naming external variables listed in Section 6.1. No constant name can have embedded spaces.

The types of external constants you can refer to vary from system to system. In *VAX–11 BASIC*, the named constant might be a system status code or a global constant declared in a *VAX–11* MACRO or *VAX–11* BLISS program. In *BASIC–PLUS–2*, the named constant might be a global constant declared in a MACRO–11 program or an RMS constant. See the user's guide for your system for more information on external constants available to your programs.

5.4 Explicit Literal Notation

You can specify the value and data type of numeric literals by using a special notation. The format of this notation in *VAX–11 BASIC* is:

```
[radix] num-str-lit [data-type]
```

Radix specifies an optional base.

In *VAX–11 BASIC*, radix can be:

- D Decimal (base 10)
- B Binary (base 2)
- O Octal (base 8)
- X Hexadecimal (base 16)

The *VAX–11 BASIC* default radix is D, but you can also specify binary, octal, and hexadecimal integer literals. Binary, octal, and hexadecimal notation allows you to set or clear individual bits in the representation of an integer. This feature is useful in forming conditional expressions and in using logical operations.

In *BASIC–PLUS–2*, *num-str-lit* is always treated as decimal (base 10), so the format for explicit literal notation in *BASIC–PLUS–2* is:

```
num-str-lit [data-type]
```

Num-str-lit is a quoted string that can consist of digits and an optional decimal point when the radix is decimal. You can also use E notation for floating-point constants. A leading minus sign cannot appear inside the quotation marks, but can appear before the radix.

In *VAX–11 BASIC*, *num-str-lit* can be the digits 0 and 1 when the radix is binary, the digits 0 through 7 when the radix is octal, and the digits 0 through F when the radix is hexadecimal.

Data-type is an optional single letter that corresponds to a data-type keyword, excluding INTEGER and REAL:

- B BYTE
- W WORD
- L LONG
- F SINGLE

(continued on next page)

- D DOUBLE
- G GFLOAT (*VAX–11 BASIC* only)
- H HFLOAT (*VAX–11 BASIC* only)
- P DECIMAL (*VAX–11 BASIC* only)

For example:

"255"L	Specifies a LONG decimal constant with a value of 255.
"4000"F	Specifies a SINGLE decimal constant with a value of 4000.
–"125"B	Specifies a BYTE decimal constant with a value of –125.

A quoted numeric string alone, without a *radix* and a *data-type*, is a string literal, not a numeric literal. For example:

"255"W	Specifies a WORD decimal constant with a value of 255.
"255"	Is a string literal.

In *VAX–11 BASIC*, if you specify a binary, octal, or hexadecimal *radix*, *data-type* must be an integer. If you do not specify a data type, BASIC uses the default integer data type. For example:

B"11111111"B	Specifies a BYTE binary constant with a value of –1.
B"11111111"W	Specifies a WORD binary constant with a value of 255.
B"11111111"	Specifies a binary constant of the default data type (BYTE, WORD, or LONG).
B"11111111"F	Is illegal because F is not an integer data type.
X"FF"B	Specifies a BYTE hexadecimal constant with a value of –1.
X"FF"W	Specifies a WORD hexadecimal constant with a value of 255.
X"FF"D	Is illegal because D is not an integer data type.
O"377"B	Specifies a BYTE octal constant with a value of –1.
O"377"W	Specifies a WORD octal constant with a value of 255.
O"377"G	Is illegal because G is not an integer data type.

When you specify a radix other than decimal, *VAX–11 BASIC* treats the numeric string as an unsigned integer. When, however, this value is assigned to a variable or used in an expression, *VAX–11 BASIC* treats the variable as a signed integer. For example:

```
100     DECLARE BYTE A
        A = B"11111111"B
        PRINT A
RUNNH

-1
```

In this example, *VAX–11 BASIC* sets all eight bits in storage location A. Because A is a BYTE integer, it has only 8 bits of storage and its value is –1 (the 8–bit two's complement of 1 is 11111111). If the data type were W (WORD), *VAX–11 BASIC* would set the bits to 0000000011111111, and its value would be 255.

Note that in *VAX–11 BASIC* a D can appear in both the radix position and the data type position. D in the radix position specifies that the numeric string is to be treated as a decimal number (base 10). D in the data type position specifies that the value is to be treated as a double-precision, floating-point constant. A P in the data type position specifies a packed decimal constant. For example:

"255"D Specifies a double-precision constant with a value of 255.

"255.55"P Specifies a DECIMAL constant with a value of 255.55.

You can also use explicit literal notation to represent a single-character string in terms of its 8–bit ASCII value. The format in *VAX–11 BASIC* is:

[radix] num-str-lit C

The format in *BASIC–PLUS–2* is:

num-str-lit C

The letter C is an abbreviation for CHARACTER. The value of the numeric string must be between 0 and 255, inclusive.

This feature lets you create your own compile-time string constants containing nonprinting characters. For example:

```
100     DECLARE STRING CONSTANT CONTROL_G = "7"C
        PRINT CONTROL_G
```

This example declares a string constant named CONTROL_G (ASCII decimal value 7). When BASIC executes the PRINT statement, the terminal bell sounds.

See the *BASIC User's Guide* for more information on explicit literal notation.

5.5 Predefined Constants

Predefined constants are symbolic representations of either: 1) ASCII characters or 2) mathematical values. They are also called compile-time constants because their value is known at compile time rather than at run time. Predefined constants:

- Format program output to improve readability
- Make source code easier to understand

Table 4 lists predefined constants supplied by BASIC, their ASCII values, and their purposes.

Table 4: Predefined Constants

Constant	Decimal ASCII Value	Purpose
BEL (Bell)	7	Sounds the terminal bell
BS (Backspace)	8	Moves the cursor one position to the left
HT (Horizontal Tab)	9	Moves the cursor to the next horizontal tab stop

(continued on next page)

Table 4: Predefined Constants (Cont.)

Constant	Decimal ASCII Value	Purpose
LF (Line Feed)	10	Moves the cursor to the next line
VT (Vertical Tab)	11	Moves the cursor to the next vertical tab stop
FF (Form Feed)	12	Moves the cursor to the start of the next page
CR (Carriage Return)	13	Moves the cursor to the beginning of the current line
SO (Shift Out)	14	Shifts out for communications networking, screen formatting, and alternate graphics
SI (Shift In)	15	Shifts in for communications networking, screen formatting, and alternate graphics
ESC (Escape)	27	Marks the beginning of an escape sequence
SP (Space)	32	Inserts one blank space in program output
DEL (Delete)	127	Deletes the last character entered
PI	None	Represents the number PI with the precision of the default floating-point data type

You can use predefined constants in many ways. For example, to print and underline a word on a hard copy terminal:

```
110     PRINT "NAME:" + BS + BS + BS + BS + BS + "_____"
120     END

RUNNH

NAME:
```

To print and underline a word on a VT100 video display terminal:

```
100     PRINT ESC + "[4mNAME:" + ESC + "[0m"
110     END

RUNNH

NAME:
```

Note that the "m" in the above example must be lowercase.

You can also create your own predefined constants with the DECLARE CONSTANT statement. For example:

```
10      DECLARE STRING CONSTANT Underlined_name = ESC + "[4mNAME:" + ESC + "[0m"
20      DECLARE DOUBLE CONSTANT D_PI = PI
30      PRINT Underlined_name
        PRINT D_PI,,PI
```

Line 10 defines Underlined_name as a string constant equivalent to the constant displayed by line 100 in the previous example. Line 20 defines D_PI as a DOUBLE constant equal to the predefined constant PI. If the default REAL data size is SINGLE, the program can use both single-precision PI and double-precision D_PI. See the *BASIC User's Guide* for more information on predefined constants and their use in BASIC programs.

6.0 Variables

A variable is a named quantity whose value can change during program execution. Each variable name refers to a location in the program's storage area. Each location can hold only one value at a time. Variables of all data types can have subscripts that indicate their position in an array.

Depending on the program operations specified, the value of a variable can change from statement to statement. BASIC uses the most recently assigned value when performing calculations. This value remains until another statement assigns a new value to the variable.

You can declare variables implicitly or explicitly.

BASIC accepts these general types of variables:

- Floating-point
- Integer
- String
- RFA
- Packed Decimal (*VAX–11 BASIC* only)
- Record (*VAX–11 BASIC* only)

See Chapter 9 in the *BASIC User's Guide* for more information on RFA variables and Chapter 6 in *BASIC on VAX/VMS Systems* for more information on record data structures.

6.1 Variable Names

The name given to a variable depends on whether the variable is internal or external to the program and whether the variable is implicitly or explicitly declared.

1. The name of an internal, explicitly declared variable must conform to the following rules:
   - The name consists of from 1 to 31 characters.
   - The first character of the name must be an upper- or lowercase alphabetic character (A through Z).
   - The last character of the name cannot be a dollar sign ($) or a percent sign (%).
   - The remaining characters, if present, can be any combination of upper- or lowercase letters (A through Z), numbers (0 through 9), dollar signs ($), underscores (_), or periods (.). The use of underscores in variable names helps improve readability and is preferred to the use of periods.
2. The name of an internal, implicitly declared variable must conform to the following rules:
   - The name consists of from 1 to 31 characters.
   - The first character of the name must be an upper- or lowercase alphabetic character (A through Z).
   - The last character of the name can be either a dollar sign ($) to indicate a string variable or a percent sign (%) to indicate an integer variable. If the last character is neither a dollar sign nor a percent sign, the name indicates a variable of the default type.

- The remaining characters, if present, can be any combination of upper- or lowercase letters (A through Z), numbers (0 through 9), dollar signs ($), underscores (_), or periods (.). The use of underscores in variable names helps improve readability and is preferred to the use of periods.

3. The name of an external, explicitly declared variable in *VAX–11 BASIC* must follow the rules for naming an internal, explicitly declared variable.

4. The name of an external, explicitly declared variable in *BASIC–PLUS–2* must conform to the following rules:

   - The name consists of from one to six characters.
   - The first character of the name must be an upper- or lowercase alphabetic character (A through Z).
   - The remaining characters, if present, can be any combination of upper- or lowercase letters (A through Z), numbers (0 through 9), dollar signs ($), or periods (.).

5. A program cannot have external, implicitly declared variables since all implicitly declared names except SUB subprogram names are internal to the program.

In all cases, no variable name can have embedded spaces.

6.2 Implicitly Declared Variables

BASIC accepts three types of implicitly declared variables:

- Floating-point (or default data type)
- Integer
- String

The name of an implicitly declared variable defines its data type. Integer variables end with a percent sign (%), string variables end with a dollar sign ($), and variables of the default type (usually floating-point) end with any allowable character except a percent sign or dollar sign. All three types of variables must conform to the rules listed in Section 6.1 for naming variables. The current data-type default (INTEGER, REAL, or, in *VAX–11 BASIC*, DECIMAL) determines the data type of implicitly declared variables that do not end in a percent sign (%) or dollar sign ($).

A floating-point variable is a named location that stores a single floating-point value. The current default size for floating-point numbers (SINGLE, DOUBLE, or, in *VAX–11 BASIC*, GFLOAT or HFLOAT) determines the data type of the floating-point variable. The following are valid floating-point variable names:

```
C          L...5        ID_NUMBER
M1         BIG47        STORAGE.LOCATION.FOR.XX
F67T.J     Z2.          STRESS_VALUE
```

If a numeric value of a different data type is assigned to a floating-point variable, BASIC converts the value to a floating-point number.

An integer variable is a named location that stores a single integer value. The current default size for integers (BYTE, WORD, or LONG) determines the data type of an integer variable. The following are valid integer variable names:

ABCDEFG%	C_8%	RECORD.NUMBER%
B%	D6E7%	THE.VALUE.I.WANT%

If the default data type is INTEGER, the percent suffix (%) is not necessary.

If you assign a floating-point or decimal (*VAX–11 BASIC* only) value to an integer variable, BASIC truncates the fractional portion of the value. It does not round to the nearest integer. For example:

```
100     B% = -5.7
```

BASIC assigns the value –5 to the integer variable, not –6.

A string variable is a named location that stores strings. The following are valid string variable names:

C1$	M$	EMPLOYEE_NAME$
L.6$	F34G$	TARGET.RECORD$
ABC1$	T..$	STORAGE_SHELF_IDENTIFIER$

Strings have both value and length. BASIC sets all string variables to a default length of zero before program execution begins, except those in a COMMON, MAP, or virtual array. See Sections 5.0 and 35.0 in Part IV of this manual for information on string length in COMMON and MAP areas. See the *BASIC User's Guide* for information on default string length in virtual arrays.

During execution, the length of a character string associated with a string variable can vary from zero (signifying a null or empty string) to 65535 characters in *VAX–11 BASIC* or 32767 characters in *BASIC–PLUS–2*.

6.3 Explicitly Declared Variables

In addition to implicitly declared variables described in the previous sections, BASIC lets you explicitly assign a data type to a variable or an array. For example:

```
100     DECLARE DOUBLE Interest_rate
```

Data-type keywords are described in Section 4.0. For more information on explicit declaration of variables, see the sections on COMMON, DECLARE, DIMENSION, DEF, FUNCTION, EXTERNAL, MAP, and SUB in Part IV of this manual and Chapter 5 in the *BASIC User's Guide*.

6.4 Subscripted Variables and Arrays

A subscripted variable is part of an array. Arrays can be of any valid data type. Subscripted variables and arrays follow the same naming conventions as nonsubscripted variables. Subscripts follow the variable name in parentheses and define the variable's position in the array. When you create an array, bounds follow the array name in parentheses and define the maximum size of the array. For example:

```
100     DECLARE STRING Emp_name(1000)
200     FOR I% = 0% TO 1000%
                INPUT "Employee name";Emp_name(I%)
        NEXT I%
```

The DECLARE statement in the example on the previous page sets the bounds of array Emp_name to 1000. Thus, the maximum value for an Emp_name subscript is 1000. The bounds of the array define the maximum value for a subscript of that array.

In *VAX–11 BASIC*, subscripts can be any positive integer value from 0 to 2147483646 in LONG mode. In *BASIC–PLUS–2*, subscripts can be any non-negative integer value from 0 to 32766.

Note

> The compiler signals an error if a subscript is bigger than the allowable range. Also, the amount of storage the system can allocate depends on available memory. Therefore, very large arrays may cause an internal allocation error.

An array is a set of data ordered in any number of dimensions. A one-dimensional array, like Emp_name(1000), is called a list or vector. A two-dimensional array, like Payroll_data(5,5), is called a matrix. An array of more than two dimensions, like Big_array(15,9,2), is called a tensor.

BASIC arrays are always zero-based. That is, the number of elements in any dimension always includes element number zero. For example, the array Emp_name(1000) contains 1001 elements, since BASIC allocates element zero. Payroll_data(5,5) contains 36 elements because BASIC always allocates row and column zero.

For all arrays except virtual arrays, the total number of array elements cannot exceed 2147483647 in *VAX–11 BASIC* and 32767 in *BASIC–PLUS–2*. For example, *VAX–11 BASIC* allows array A(2147483646) but does not allow array A(1,2147483646). *BASIC–PLUS–2* allows array A(32766) but does not allow array A(1,32766).

VAX–11 BASIC arrays can have up to 32 dimensions. *BASIC–PLUS–2* arrays can have up to eight dimensions. You can also specify the type of data the array contains with data-type keywords. Table 2 lists BASIC data types.

An element in a one-dimensional array has a variable name followed by one subscript in parentheses. There can be a space between the array name and the parenthetical subscripts. For example:

```
A(6%)

B (6%)

C$ (6%)
```

A(6%) refers to the seventh item in this list:

```
A(0%)    A(1%)    A(2%)    A(3%)    A(4%)    A(5%)    A(6%)
```

An element in a two-dimensional array has two subscripts, in parentheses, following the variable name. The first subscript specifies the row number, the second specifies the column. Use a comma to separate the subscripts. There can be a space between the array name and the parenthetical subscripts. For example:

```
A (7%,2%)    A%(4%,6%)    A$(10%,10%)
```

In the following table, the arrow points to the element specified by the subscripted variable A%(4%,6%):

		C	O	L	U	M	N	S	
		0	1	2	3	4	5	6	
R	0	0	0	0	0	0	0	0	
O	1	0	0	0	0	0	0	0	
W	2	0	0	0	0	0	0	0	
S	3	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	← A%(4%,6%)

An element in an array has as many subscripts as there are dimensions. An element of Big_array(15%,9%,2%), for example, would have three subscripts.

Although a program can contain a variable and an array with the same name, this is regarded as poor programming practice. Variable A and the array A(3%,3%) are separate entities and are stored in completely separate locations and should have different names.

Note

A program cannot contain two arrays with the same name and a different number of subscripts. For example, the arrays A(3%) and A(3%,3%) are invalid in the same program.

BASIC arrays can be redimensioned at run time. See Chapter 7 in the *BASIC User's Guide* for more information on arrays.

6.5 Initialization of Variables

BASIC sets variables to zero or null values at the start of program execution. Variables initialized by BASIC include:

- Numeric variables and in-storage array elements (except those in MAP or COMMON statements).
- String variables (except those in MAP or COMMON statements).
- Local variables in function definitions. In addition, BASIC sets these values to zero each time the program calls the function.
- Variables in subprograms. Subprogram variables are initialized to zero or the null string each time the subprogram is called.

BASIC does not initialize virtual arrays.

Note

In *BASIC–PLUS–2*, variables in a MAP statement referenced in an OPEN statement are initialized to zero or the null string when the file is opened. In *VAX–11 BASIC*, these variables are not initialized. You can also use MACRO–11 routines to initialize MAP and COMMON areas. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information.

7.0 Expressions

BASIC expressions consist of operands (numbers, strings, constants, variables, functions, or array elements) separated by:

- Arithmetic operators
- String operators
- Relational operators
- Logical operators

All BASIC expressions except string concatenation and invocations of string-valued functions yield numeric values. The way you combine numeric operators and operands and use the resulting values allows you to produce:

- Numeric expressions
- String expressions
- Conditional expressions

BASIC evaluates expressions according to operator precedence and uses the results in program execution. Parentheses can appear in expressions to group operands and operators, thus controlling the order of evaluation.

The following sections explain the types of expressions you can create and the way BASIC evaluates expressions.

7.1 Numeric Expressions

Numeric expressions consist of floating-point, integer, or packed decimal (*VAX–11 BASIC* only) operands separated by arithmetic operators and optionally grouped by parentheses. Table 5 shows how numeric operators work in numeric expressions.

Table 5: Arithmetic Operators

Operator	Example	Use
+	A + B	Add B to A
–	A – B	Subtract B from A
*	A * B	Multiply A by B
/	A / B	Divide A by B
^	A^B	Raise A to the power B
**	A**B	Raise A to the power B

In general, two arithmetic operators cannot occur consecutively in the same expression. Exceptions are the unary plus and unary minus. The following expressions are valid:

A * + B

A * – B

A * (–B)

A * + – + – B

The following expression is not valid:

```
A - * B
```

An operation on two numeric operands of the same data type yields a result of that type. For example:

A% + B% yields an integer value of the default type.

G3 * M5 yields a floating-point value if the default type is REAL.

If the result of the operation exceeds the range of the data type, *VAX–11 BASIC* signals an overflow error message. For example:

```
10      DECLARE BYTE A, B
        A = 127
        B = 127
        PRINT A + B
99      END
```

This example causes *VAX–11 BASIC* to signal the error "Integer error or overflow" because the sum of A and B (254) exceeds the range of –128 to +127 for BYTE integers. Similar overflow errors occur for REAL and DECIMAL data types whenever the result of a numeric operation is outside the range of the data type.

Assigning a value of one data type to a variable of a different data type changes the assigned value's data type to the variable's data type. For example:

```
10      A% = 5.1 * 6.3
```

This example assigns the value 32 to the integer variable A% even though the floating-point value of the expression is 32.13. This is called numeric conversion. See Chapter 5 of the *BASIC User's Guide* for more information on numeric conversion.

7.1.1 Floating-Point and Integer Promotion Rules

When an expression contains operands with different data types, the data type of the result is determined by BASIC's data type promotion rules:

- With one exception, BASIC promotes operands with different data types to the lowest common data type that can hold the largest or most precise possible value of either operand's data type, then performs the operation in that data type, and yields a result of that data type.
- The exception to the previous rule is that when an operation involves SINGLE and LONG data types, BASIC promotes the LONG data type to SINGLE, rather than to DOUBLE, performs the operation, and yields a result of the SINGLE data type.

Note that BASIC does a sign extend when converting BYTE and WORD integers to a higher INTEGER data type (WORD or LONG). That is, the high order bit (the sign bit) determines how the additional bits are set when the BYTE or WORD is converted to WORD or LONG. If the high order bit is zero (positive), all higher-order bits in the converted BYTE or WORD are set to zero. If the high order bit is one (negative), all higher-order bits in the converted BYTE or WORD are set to one.

Table 6 lists the data type results possible in numeric expressions that combine BYTE, WORD, LONG, SINGLE, and DOUBLE data. Table 7 lists the data type results possible in numeric expressions that combine the *VAX–11 BASIC* only data types, GFLOAT and HFLOAT. Note that in *VAX–11 BASIC*, when the operands are DOUBLE and GFLOAT, BASIC promotes both values to HFLOAT, and

returns an HFLOAT value. The promotion of DOUBLE and GFLOAT to HFLOAT is necessary because a DOUBLE value is more precise than a GFLOAT value, but cannot contain the largest possible GFLOAT value. Consequently, BASIC promotes these data types to a data type that can hold the largest and most precise value of either operand.

Table 6: Result Data Types in BASIC Expressions

	Operand 2				
Operand 1	**BYTE**	**WORD**	**LONG**	**SINGLE**	**DOUBLE**
BYTE	BYTE	WORD	LONG	SINGLE	DOUBLE
WORD	WORD	WORD	LONG	SINGLE	DOUBLE
LONG	LONG	LONG	LONG	SINGLE	DOUBLE
SINGLE	SINGLE	SINGLE	SINGLE	SINGLE	DOUBLE
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE

VAX **Table 7: VAX–11 BASIC Result Data Types**

	Operand 2	
Operand 1	**GFLOAT**	**HFLOAT**
BYTE	GFLOAT	HFLOAT
WORD	GFLOAT	HFLOAT
LONG	GFLOAT	HFLOAT
SINGLE	GFLOAT	HFLOAT
DOUBLE	HFLOAT	HFLOAT
GFLOAT	GFLOAT	HFLOAT
HFLOAT	HFLOAT	HFLOAT

As Table 6 shows, if one operand is SINGLE and one operand is DOUBLE, BASIC promotes the SINGLE value to DOUBLE, performs the specified operation, and returns the result as a DOUBLE value. This promotion is necessary because the SINGLE data type has less precision than the DOUBLE value, whereas the DOUBLE data type can represent all possible SINGLE values. If BASIC did not promote the SINGLE value and the operation yielded a result outside of the SINGLE range, loss of precision and significance would occur.

The data types BYTE, WORD, LONG, SINGLE, and DOUBLE form a simple hierarchy: if all operands in an expression are these data types, the result of the expression is the highest data type used in the expression.

VAX **7.1.2 DECIMAL Promotion Rules (VAX–11 BASIC only)**

VAX–11 BASIC also allows the DECIMAL(d,s) data type. The number of digits (d) and the scale or position of the decimal point (s) in the result of operations involving a DECIMAL value depends on the

data type of the other operand. If one operand is DECIMAL and the other is DECIMAL or INTEGER, the d and s values of the result are determined as follows:

- If both operands are typed DECIMAL, and if both operands have the same digit (d) and scale (s) values, no conversions occur and the result of the operation has exactly the same d and s values as the operands. Note, however, that overflow can occur if the result exceeds the range specified by the d value.

- If both operands are DECIMAL but have different digit and scale values, BASIC always uses the larger number of specified digits for the result.

  For example:

```
100     DECLARE DECIMAL(5,2) A
        DECLARE DECIMAL(4,3) B
```

  Variable A allows three digits to the left of the decimal point and two digits to the right. Variable B allows one digit to the left of the decimal point and three digits to the right. Therefore, the result allows three digits to the left of the decimal point and three digits to the right:

```
A          ---.--
B            -.---

Result     ---.---
```

- If one operand is typed DECIMAL and one is typed INTEGER, the INTEGER value is converted to a DECIMAL(d,s) data type as follows:

  BYTE is converted to DECIMAL(3,0).

  WORD is converted to DECIMAL(5,0).

  LONG is converted to DECIMAL(10,0).

  BASIC then determines the d and s values of the result by evaluating the d and s values of the operands as described above.

Note that only INTEGER data types are converted to the DECIMAL data type. If one operand is DECIMAL and one is floating-point, the DECIMAL value is converted to a floating-point value. The total number of digits (d) in the DECIMAL value determines its new data type, as shown in Table 8.

Table 8: Result Data Types for DECIMAL Data

Number of DECIMAL Digits in Operand	Floating-point Operands			
	SINGLE	**DOUBLE**	**GFLOAT**	**HFLOAT**
1–6	SINGLE	DOUBLE	GFLOAT	HFLOAT
7–15	DOUBLE	DOUBLE	GFLOAT	HFLOAT
16	DOUBLE	DOUBLE	HFLOAT	HFLOAT
17–31	HFLOAT	HFLOAT	HFLOAT	HFLOAT

If the value of d is between 7 and 15, the operand is converted to:

- DOUBLE if the floating-point operand is SINGLE or DOUBLE
- GFLOAT if the floating-point operand is GFLOAT
- HFLOAT if the floating-point operand is HFLOAT

Thus, a DECIMAL(8,5) operand is converted to DOUBLE if the other operand is SINGLE or DOUBLE, to GFLOAT if the other operand is GFLOAT, and to HFLOAT if the other operand is HFLOAT.

Note also that exponentiation of a DECIMAL data type returns a REAL value.

See the *BASIC User's Guide* for more information on data type interactions, conversions, and promotion rules in BASIC numeric expressions.

7.2 String Expressions

String expressions are string entities separated by the plus sign (+). When used in a string expression, the plus sign concatenates strings.

For example:

```
100     INPUT "Type two words to be combined";A$, B$
        C$ = A$ + B$
        PRINT C$
200     END

RUNNH

Type two words to be combined? hello
? goodbye

hellogoodbye

Ready
```

7.3 Conditional Expressions

Conditional expressions can be either relational or logical expressions.

Numeric relational expressions compare numeric operands to determine whether the expression is true or false. String relational expressions compare string operands to determine which string expression occurs first in the ASCII collating sequence.

Logical expressions contain integer operands and logical operators. BASIC determines whether the specified logical expression is true or false by testing the numeric result of the expression. Note that in conditional expressions, as in any numeric expression, when BYTE and WORD operands are converted to WORD and LONG, the specified operation is performed in the higher data type, and the result returned is also of the higher data type. When one of the operands is a negative value, this conversion will produce accurate but perhaps confusing results, because BASIC performs a sign extend when converting BYTE and WORD integers to a higher integer data type. See Section 7.1.1 for information on integer conversion rules.

7.3.1 Numeric Relational Expressions

Operators in numeric relational expressions compare the values of two operands and return: 1) a minus one if the relation is true or 2) a zero if the relation is false. The data type of the result is the default integer type. For example:

Example 1

```
100     A = 10
        B = 15
        X% = (A <> B)
        IF X% = -1%
        THEN PRINT 'Relationship is true'
        ELSE IF X% = 0
             THEN PRINT 'Relationship is false'
             END IF
        END IF

RUNNH

Relationship is true
```

Example 2

```
10 A = 10
   B = 15
   X% = A = B
   IF X% = -1%
   THEN PRINT 'Relationship is true'
   ELSE IF X% = 0
        THEN PRINT 'Relationship is false'
        END IF
   END IF

RUNNH

Relationship is false
```

Table 9 shows how numeric operators work in numeric relational expressions.

Table 9: Numeric Relational Operators

Operator	Example	Meaning
=	A = B	A is equal to B.
<	A < B	A is less than B.
>	A > B	A is greater than B.
<= or =<	A <= B	A is less than or equal to B.
>= or =>	A >= B	A is greater than or equal to B.
<> or ><	A <> B	A is not equal to B.
==	A == B	A and B will PRINT the same because they are equal to six significant digits.

7.3.2 String Relational Expressions

Operators in string relational expressions determine how BASIC compares strings. BASIC determines the value of each character in the string by converting it to its ASCII value. ASCII values are listed in Appendix C in *BASIC on VAX/VMS Systems*, *BASIC on RSX–11M/M–PLUS Systems*, and *BASIC on RSTS/E Systems*. BASIC compares the strings character by character, left to right, until it finds a difference in ASCII value. For example:

```
10 A$ = 'ABC'
   B$ = 'ABZ'
20 IF A$ < B$
   THEN PRINT 'ABC comes before ABZ'
        GOTO 99
   ELSE IF A$ == B$
        THEN PRINT 'The strings are identical'
             GOTO 99
        ELSE IF A$ > B$
             THEN PRINT 'ABC comes after ABZ'
                  GOTO 99
             END IF
        END IF
   END IF
55 PRINT 'Strings are equal but not identical'
99 END
```

In this example, BASIC compares A$ and B$ character by character. The strings are identical up to the third character. Because the ASCII value of "Z" (90) is greater than the ASCII value of "C" (67), A$ is less than B$. BASIC evaluates the expression A$ < B$ as true (–1), prints "ABC comes before ABZ" and goes to line 99.

If two strings of differing lengths are identical up to the last character in the shorter string, BASIC pads the shorter string with spaces (ASCII value 32) to generate strings of equal length, unless the operator is the double equals sign (= =). If the operator is the double equals sign, BASIC does not pad the shorter string. For example:

```
10 A$ = 'ABCDE'
   B$ = 'ABC'
20 PRINT 'B$ comes before A$' IF B$ < A$
   PRINT 'A$ comes before B$' IF A$ < B$
30 C$ = 'ABC '
   IF B$ == C$
         THEN PRINT 'B$ exactly matches C$'
         ELSE PRINT 'B$ does not exactly match C$'
   END IF
   IF B$ = C$
         THEN PRINT 'B$ matches C$ with padding'
         ELSE PRINT 'B$ does not match C$'
   END IF

RUNNH

B$ comes before A$
B$ does not exactly match C$
B$ matches C$ with padding
```

In this program, BASIC compares "ABCDE" to "ABC " to determine which string comes first in the collating sequence. "ABC " comes before "ABCDE" because the ASCII value for space (32) is lower than the ASCII value of "D" (68). Then BASIC compares "ABC " with "ABC" using the double

equals sign and determines that the strings do not match exactly without padding. The third comparison uses the single equals sign. BASIC pads "ABC" with spaces and determines that the two strings match with padding.

Table 10 shows how numeric operators work in string relational expressions.

Table 10: String Relational Operators

Operator	Example	Meaning
=	A$ = B$	Strings A$ and B$ are identical after the shorter string has been padded with spaces to equal the length of the longer string.
<	A$ < B$	String A$ occurs before string B$ in ASCII sequence.
>	A$ > B$	String A$ occurs after string B$ in ASCII sequence.
<= or =<	A$ <= B$	String A$ is identical to or precedes string B$ in ASCII sequence.
>= or =>	A$ >= B$	String A$ is identical to or follows string B$ in ASCII sequence.
<> or ><	A$ <> B$	String A$ is not identical to string B$.
==	A$ == B$	Strings A$ and B$ are identical in composition and length, without padding.

BASIC treats unquoted strings typed in response to the INPUT statement differently than quoted strings by ignoring leading and trailing spaces and tabs. That is, BASIC evaluates the quoted strings "ABC" and "ABC " as equal but not identical because the == operator does not pad the shorter string with spaces. When you input the same strings as unquoted strings in response to the INPUT prompt, BASIC evaluates them as equal and identical because it ignores the trailing spaces. The LINPUT statement, on the other hand, treats unquoted strings as string literals so the trailing spaces are part of the string, and BASIC evaluates the strings as equal but not identical.

7.3.3 Logical Expressions

A logical expression contains either:

- A unary logical operator and one integer operand
- Two integer operands separated by a binary logical operator
- One integer operand

Logical expressions are valid only when the operands are integers. If the expression contains two integer operands of differing data types, the resulting integer has the same data type as the higher integer operand. For instance, the result of an expression that contains a BYTE integer and a WORD integer would be a WORD integer. Table 6 shows how integer data types interact with each other in expressions.

BASIC determines whether the condition is true or false by testing the result of the logical expression to see whether any bits are set. If no bits are set, the value of the expression is zero and it is evaluated as false; if any bits are set, the value of the expression is nonzero, and the expression is evaluated as true. BASIC generally accepts any nonzero value in logical expressions as true. However, logical operators can return unanticipated results unless minus one is specified for true values and zero for false. Therefore, logical operators should be used on the results of relational expressions to obtain valid and predictable results. Table 11 lists logical operators. Examples that show how logical operators work on nonzero and minus one values follow the table.

Table 11: Logical Operators

Operator	Example	Meaning
NOT	NOT A%	The bit-by-bit complement of A%. If A% is true (–1), NOT A% is false (0).
AND	A% AND B%	The logical product of A% and B%. A% AND B% is true only if both A% and B% are true.
OR	A% OR B%	The logical sum of A% and B%. A% OR B% is false only if both A% and B% are false; otherwise, A% OR B% is true.
XOR	A% XOR B%	The logical exclusive OR of A% and B%. A% XOR B% is true if either A% or B% is true but not if both are true.
EQV	A% EQV B%	The logical equivalence of A% and B%. A% EQV B% is true if A% and B% are both true or both false; otherwise, the value is false.
IMP	A% IMP B%	The logical implication of A% and B%. A% IMP B% is false only if A% is true and B% is false; otherwise, the value is true.

The truth tables in Table 12 summarize the results of these logical operations. Zero is false; minus one is true.

Table 12: Truth Tables

A%	NOT A%
0	–1
–1	0

A%	B%	A% AND B%
0	0	0
0	–1	0
–1	0	0
–1	–1	–1

A%	B%	A% XOR B%
0	0	0
0	–1	–1
–1	0	–1
–1	–1	0

A%	B%	A% OR B%
0	0	0
0	–1	–1
–1	–0	–1
–1	–1	–1

A%	B%	A% EQV B%
0	0	–1
0	–1	0
–1	0	0
–1	–1	–1

A%	B%	A% IMP B%
0	0	–1
0	–1	–1
–1	0	0
–1	–1	–1

The operators XOR and EQV are logical complements.

Note that in logical expressions, any nonzero value is evaluated as true, while in relational expressions, a minus one is generated as a true value. Logical operators set bits in the result of the expression; any bit set is a nonzero value and is evaluated as true. For this reason, it is important to use logical operators on the results of relational expressions (the values of minus one and zero) to avoid unanticipated results. For example:

```
10   A% = 2%
20   B% = 4%
30   IF A% THEN PRINT 'A% IS TRUE'
```

```
40    IF B% THEN PRINT 'B% IS TRUE'
50    IF A% AND B% THEN PRINT 'A% AND B% IS TRUE'
                    ELSE PRINT 'A% AND B% IS FALSE'
60    END

RUNNH

A% IS TRUE
B% IS TRUE
A% AND B% IS FALSE
```

In this example, the values of A% and B% both test as true because they are nonzero values. However, the logical AND of these two variables returns an unanticipated result of "false."

The program returns this seemingly contradictory result because logical operators work on the individual bits of the operands. The 8–bit binary representation of 2% is:

0 0 0 0 0 0 1 0

The 8–bit binary representation of 4% is:

0 0 0 0 0 1 0 0

Each value tests as true because it is nonzero. However, the AND operation on these two values sets a bit in the result only if the corresponding bit is set in both operands. Therefore, the result of the AND operation on 4% and 2% is:

0 0 0 0 0 0 0 0

No bits are set in the result, so the value tests as false (zero).

If the value of B% is changed to 6%, the resulting value tests as true (nonzero) because both 6% and 2% have the second bit set. Therefore, BASIC sets the second bit in the result and the value tests as nonzero and true.

The 8–bit binary representation of minus one is:

1 1 1 1 1 1 1 1

The result of –1% AND –1% is –1% because BASIC sets bits in the result for each corresponding bit that is set in the operands. The result, therefore, tests as true because it is a nonzero value. For example:

```
10    A% = -1%
20    B% = -1%
30    IF A% THEN PRINT 'A% IS TRUE'
40    IF B% THEN PRINT 'B% IS TRUE'
50    IF A% AND B% THEN PRINT 'A% AND B% IS TRUE'
                    ELSE PRINT 'A% AND B% IS FALSE'
60    END

RUNNH

A% IS TRUE
B% IS TRUE
A% AND B% IS TRUE
```

Your program may also return unanticipated results if you use the NOT operator with a nonzero operand that is not minus one. For example:

```
10    A%=-1%
20    B%=2
30    IF A% THEN PRINT 'A% IS TRUE'
            ELSE PRINT 'A% IS FALSE'
40    IF B% THEN PRINT 'B% IS TRUE'
            ELSE PRINT 'B% IS FALSE'
50    IF NOT A% THEN PRINT 'NOT A% IS TRUE'
                  ELSE PRINT 'NOT A% IS FALSE'
60    IF NOT B% THEN PRINT 'NOT B% IS TRUE'
                  ELSE PRINT 'NOT B% IS FALSE'
99    END

RUNNH

A% IS TRUE
B% IS TRUE
NOT A% IS FALSE
NOT B% IS TRUE
```

In this example, BASIC evaluates both A% and B% as true because they are nonzero. NOT A% is evaluated as false (zero) because the binary complement of minus one is zero. NOT B% is evaluated as true because the binary complement of two has bits set and, therefore, is a nonzero value.

Note

DIGITAL recommends that you use logical operators on the results of relational expressions to avoid obtaining unanticipated results.

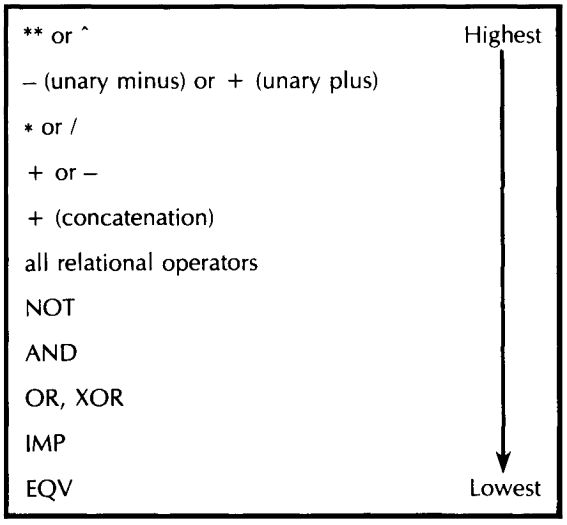
7.4 Evaluating Expressions

BASIC evaluates expressions according to operator precedence. Each arithmetic, relational, and string operator in an expression has a position in the hierarchy of operators. The operator's position tells BASIC when to perform the operation. Parentheses can change the order of precedence.

Table 13 lists all operators as BASIC evaluates them. Note that:

- Operators with equal precedence are evaluated logically from left-to-right.
- BASIC evaluates expressions enclosed in parentheses first, even when the operator in parentheses has a lower precedence than that outside the parentheses.

Table 13: Numeric Operator Precedence



BASIC thus evaluates the expression A = 15^2 + 12^2 – (35 * 8) in five steps:

1.	15^2 = 225	Exponentiation (left-most expression)
2.	12^2 = 144	Exponentiation
3.	225 + 144 = 369	Addition
4.	(35 * 8) = 280	Multiplication
5.	369 – 280 = 89	Subtraction

There is one exception to this order of precedence: when an operator that does not require operands on either side of it (such as NOT) immediately follows an operator that does require operands on both sides (such as +), BASIC evaluates the second operator first. For example:

```
A% + NOT B% + C%
```

This expression is evaluated as:

```
(A% + (NOT B%)) + C%
```

BASIC evaluates the expression NOT B before it evaluates the expression A + NOT B. When the NOT expression does not follow the + expression, the normal order of precedence is followed:

```
NOT A% + B% + C%
```

This expression is evaluated as:

```
NOT ((A% + B%) + C %)
```

BASIC evaluates the two plus expressions (A% + B%) and ((A% + B%) + C%) because the plus (+) operator has a higher precedence than the NOT operator.

BASIC evaluates nested parenthetical expressions from the inside out. For example:

```
100     A = ((((25 + 5) / 5) * 7) + 3)
        PRINT A
300     B = 25 + 5 / 5 * 7 + 3
        PRINT B

RUNNH

 45
 35
```

In this program, BASIC evaluates the parenthetical expression A quite differently from expression B. For expression A, BASIC evaluates the innermost parenthetical expression (25 + 5) first, then the second inner expression (30 / 5), then (6 * 7), and finally (42 + 3). For expression B, BASIC evaluates (5 / 5) first, then (1 * 7), then (25 + 7 + 3) to obtain a different value.

PART II
Compiler Commands

1.0 APPEND

Function

The APPEND command merges an existing BASIC source program with the program currently in memory.

Format

```
APPEND [ file-spec ]
```

Syntax Rules

1. *File-spec* names the file of BASIC program lines you want to merge with the program currently in memory. The *VAX–11 BASIC* default file type is BAS, and the *BASIC–PLUS–2* default file type is B2S.

General Rules

1. If you type APPEND without specifying a file name, BASIC prompts with:

   ```
   Append file name--
   ```

   Respond with a file name. If you respond with a carriage return and no file name, *VAX–11 BASIC* searches for a file named NONAME.BAS. *BASIC–PLUS–2* searches for a file named NONAME.B2S. If the compiler cannot find NONAME.BAS or NONAME.B2S, *VAX–11 BASIC* signals the error "file not found"; *BASIC–PLUS–2* signals "can't find file or account".

2. You can append the contents of *file-spec* to a source program called into memory with the OLD command or created in the BASIC environment. If there is no program in memory, BASIC appends the *file-spec* to an empty program with the default file name, NONAME.
3. If the *file-spec* contains a BASIC line with the same line number as a line of the program in memory, the line in the appended file replaces the line of the program in memory. Otherwise, BASIC inserts appended lines into the program in memory in sequential, ascending line number order.
4. The APPEND command does not change the name of the program in memory.
5. If you have not saved the appended version of the program, BASIC signals the warning "Unsaved change has been made, CTRL/Z or EXIT to exit" the first time you try to leave the BASIC environment.

Examples

```
APPEND PROGB
```

2.0 ASSIGN (VAX–11 BASIC)

Function

The ASSIGN command equates a logical name to a complete file specification, a device, or another logical name within the context of the BASIC environment.

Format

```
ASSIGN equiv-nam[:] log-nam[:]
```

Syntax Rules

1. *Equiv-nam* specifies the file specification, device, or logical name to be assigned a logical name. If you specify a physical device name, terminate it with a colon (:).
2. *Log-nam* is the 1– to 63–character logical name to be associated with *equiv-nam*. You can specify a logical name for any portion of a file specification. If the logical name translates to a device name, and will be used in place of a device name in a file specification, terminate it with a colon (:).

General Rules

1. When the logical name assignment supersedes another logical name assigned previously, BASIC displays the message "previous logical name assignment replaced".
2. If *log-nam* has more than 63 characters, BASIC signals the error "invalid logical name".
3. Logical names assigned with the ASSIGN command are placed in the process logical name table and remain there until you exit the BASIC environment.

Examples

```
ASSIGN [LEONARD.BAS] PRO:
```

3.0 BRLRES (BASIC–PLUS–2)

Function

The BRLRES command allows you to specify a memory-resident *BASIC–PLUS–2* or user-created library to be used when you task-build the program. When you use the BUILD command, *BASIC–PLUS–2* includes the specified library in the Task Builder command file. The default library for the BRLRES command is chosen by your system manager when *BASIC–PLUS–2* is installed.

Format

```
BRLRES [ lib-param ]

lib-param:  { file-spec }
            { NONE     }
```

Syntax Rules

1. If you enter the BRLRES command without a *lib-param*, BASIC prompts for one and displays the name of the current default memory-resident library.

   - *File-spec* can be a library supplied by *BASIC–PLUS–2* or a user-created library.
   - NONE tells the Task Builder not to link your task to the *BASIC–PLUS–2* default resident library. Therefore, the Task Builder links to the *BASIC–PLUS–2* object module library, BP2OTS.OLB.
   - If you type a carriage return in response to the prompt, the current default memory-resident library is used.

General Rules

1. The memory-resident libraries supplied by *BASIC–PLUS–2* are LB:[1,1]BP2RES and LB:[1,1]BP2SML on *RSX–11M/M–PLUS systems* and LB:BP2RES and LB:BP2SML on *RSTS/E systems*. LB: is a RSTS/E logical name for the library account on disk. Because memory-resident libraries are optional, your system manager can select none, one, or both when *BASIC–PLUS–2* is installed. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on *BASIC–PLUS–2* memory-resident libraries.
2. *BASIC–PLUS–2* links the specified memory-resident library to your program when you task-build the program, so you must use the BRLRES command before you use the BUILD command to include the specified library in the Task Builder command file.
3. The BRLRES library you specify is included in your Task Builder command files until you specify a new library with the BRLRES command or exit from the BASIC environment. When you exit from the environment, the original default library is restored as the default.
4. You can override the BRLRES command with the /BRLRES qualifier added to the BUILD command, but the specified library remains in effect for only one BUILD operation.

5. The Task Builder returns an error message when the requested memory-resident library is not available.
6. Consult your system manager for information about the resident libraries available to you.

Examples

RSX–11M/M–PLUS Systems

```
BRLRES LB:[1,1]BP2RES
```

RSTS/E Systems

```
BRLRES LB:BP2RES
```

4.0 BUILD (BASIC–PLUS–2)

Function

The BUILD command generates a command (CMD) file and an overlay description language (ODL) file for the Task Builder. The command file contains instructions that enable the Task Builder to link your program module(s) with libraries and other routines. The overlay description language file specifies how segments of the task-built program are overlaid when you run it.

Format

```
BUILD [ prog-nam  [ sub-nam,... ] ] [ /qualifier ]...
```

Syntax Rules

1. *Prog-nam* names the program you want to build. If you do not specify a *prog-nam*, *BASIC–PLUS–2* creates CMD and ODL files for the current program or for NONAME if there is no current program.
2. *Sub-nam* names the subprogram or subprograms you want to link to the main program. You must specify a *prog-nam* if you specify a *sub-nam*.
3. The command file takes the name of the main program and a default extension of CMD. The ODL file takes the name of the main program and a default extension of ODL.
4. */Qualifier* specifies a qualifier keyword that sets a BASIC default. Table 17 lists all *BASIC–PLUS–2* qualifiers and describes their functions.
5. The BUILD command line must fit on a single 80–character line.

General Rules

1. The BUILD command does not change the current context of the *BASIC–PLUS–2* environment.
2. The BUILD command generates the CMD and ODL files. It does not cause the Task Builder to begin operation.
3. In addition to program names and build qualifiers, the BUILD command accepts defaults from previously specified BRLRES, DSKLIB, ODLRMS, RMSRES, LIBR, and SET commands.
4. BUILD qualifiers tell the Task Builder to perform special operations on object modules when you task-build the program. You can abbreviate all qualifiers to the first three letters of the qualifier keyword.

Examples

```
BUILD MAIN,SUB1,SUB2/DUMP/REL
```

5.0 $ Command

Function

You can enter a system command while in the BASIC environment by typing a dollar sign ($) before the command. BASIC passes the command to the operating system for execution. The context of the BASIC environment and the program currently in memory do not change in *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems*. On *RSTS/E systems*, the system command executes and control returns to the default run-time system, not to *BASIC–PLUS–2*.

Format

```
$ system-command
```

Syntax Rules

1. BASIC passes *system-command* directly to your operating system without checking for validity.

General Rules

1. The terminal displays any error messages or output that the *system-command* generates.

VAX–11 BASIC VAX

1. Control returns to the BASIC environment after the *system-command* executes. The context (source file status, loaded modules, and so on) of the BASIC environment and the program currently in memory do not change unless the *system-command* causes the operating system to abort BASIC or log you out.

2. On *VAX/VMS systems,* the *system-command* you specify executes within the context of a subprocess. Consequently, commands such as the DCL SET command execute only within the subprocess and do not affect the process running BASIC.

BASIC–PLUS–2

1. On *RSX–11M/M–PLUS systems*, control returns to the BASIC environment after the *system-command* executes. The context (source file status, loaded modules, and so on) of the BASIC environment and the program currently in memory do not change unless the *system-command* causes the operating system to abort BASIC or log you out. RSX

2. On *RSTS/E systems*, the context of the environment and the program currently in memory are lost. After the system command executes, control passes to monitor level, not to *BASIC–PLUS–2*. RSTS

3. If you have made changes to the program currently in memory, *BASIC–PLUS–2* displays the message "Unsaved change has been made – type SCRATCH or REPLACE" when you enter a *system-command*.

Examples

VAX–11 BASIC

```
Ready

$SHOW PROTECTION
  SYSTEM=RWED, OWNER=RWED, GROUP=RWED, WORLD=RE

Ready
```

BASIC–PLUS–2

```
$DIR STOCK.B2S
%Unsaved change has been made - type SCRATCH or REPLACE.

BASIC2

REPLACE

BASIC2

$DIR STOCK.B2S
```

6.0 COMPILE

Function

The COMPILE command converts a BASIC source program to an object module and writes the object file to disk.

Format

```
COMPILE [ file-spec ] [ /qualifier ]...
```

Syntax Rules

1. *File-spec* specifies a name for the output file or files. If you do not provide a *file-spec*, the compiler uses the name of the program currently in memory for the file name, a default file type of OBJ for the object file, and a default file type of LIS (*VAX–11 BASIC*) or LST (*BASIC–PLUS–2*) for the listing file, if a listing file is requested. *BASIC–PLUS–2* uses a default file type of MAC for the macro source code file when a macro file is requested.

2. In *VAX–11 BASIC*, *file-spec* can precede or follow */qualifier*. In *BASIC–PLUS–2*, *file-spec* must precede the qualifiers.

3. */Qualifier* specifies a qualifier keyword that sets a BASIC default. See Section 22.0 for information on BASIC qualifiers. Table 16 lists and describes *VAX–11 BASIC* qualifiers. Table 17 lists and describes *BASIC–PLUS–2* qualifiers.

4. In cases of ambiguous or erroneous qualifiers, *VAX–11 BASIC* signals "Unknown qualifier", *BASIC–PLUS–2* signals "Illegal switch", and the program does not compile. When qualifiers conflict, BASIC compiles the program using the last specified conflicting qualifier. For example:

   ```
   COMPILE/OBJ/NOOBJ
   ```

   BASIC compiles the program currently in memory but does not create an OBJ file.

5. You can abbreviate all positive COMPILE qualifiers to the first three letters of the qualifier keyword. A negative qualifier can be abbreviated to NO and the first three letters of the qualifier keyword.

6. There must be a program in memory or the COMPILE command does not execute and BASIC does not signal an error or warning.

General Rules

1. If an object file for the program already exists in your directory, *BASIC–PLUS–2 on RSTS/E systems* overwrites it with the new object file. *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* create a new version of the OBJ file.

2. You should not specify both a file name and file type. For example:

```
COMPILE NEWOBJ.FIL/LIS/OBJ
```

- *VAX–11 BASIC* creates two versions of NEWOBJ.FIL. The first version, NEWOBJ.FIL;1, is the listing file; the second version, NEWOBJ.FIL;2, is the object file. If you specify only a file name, BASIC uses the OBJ and LIS file type defaults when creating these files.

- *BASIC–PLUS–2* creates only the object file and names it NEWOBJ.FIL.

3. Use the COMPILE/NOOBJECT command to check your program for errors without producing an object file.

Examples

```
COMPILE NEWSTRING/DOUBLE/LIST
```

7.0 CONTINUE

Function

The CONTINUE command continues program execution after BASIC executes a STOP statement or, in *VAX–11 BASIC*, encounters a CTRL/C.

Format

```
CONTINUE
```

Syntax Rules

None.

General Rules

1. In *VAX–11 BASIC*, a program stops executing in response to a STOP statement or a CTRL/C: VAX

   - You can enter immediate mode commands and resume program execution with the CONTINUE command.
   - You cannot resume program execution if you have made source code changes or additions.

2. In *BASIC–PLUS–2*, a program stops executing when BASIC executes a STOP statement and control passes to the *BASIC–PLUS–2* debugger, which prompts with a pound sign (#). Type the CONTINUE command to resume program execution. Note that if the program was executed with the RUN/DEBUG command, you can enter debugger commands before resuming program execution with the CONTINUE command. See Part VI in this manual for more information on debugger commands. BP2

Examples

VAX–11 BASIC

```
%BAS-I-STO, Stop
-BAS-I-FROLINMOD, from line 25 in module ABC
Ready

CONTINUE
```

BASIC–PLUS–2

```
Stop at line 20

#CONTINUE
```

8.0 DELETE

Function

The DELETE command removes a specified line or range of lines from the program currently in memory.

Format

```
DELETE lin-num [ sep lin-num ] ,...

sep:    { , }
        { - }
```

Syntax Rules

1. You must enter at least one line number. If you do not, DELETE has no effect in *VAX–11 BASIC*, while *BASIC–PLUS–2* signals the error "Illegal Delete command".
2. The *sep* characters allow you to delete individual lines or a block of lines.
   - If you separate line numbers with commas, BASIC deletes each specified line number.
   - If you separate line numbers with a hyphen (–), BASIC deletes the inclusive range of lines. The lower line number must come first. If it does not, DELETE has no effect in *VAX–11 BASIC*, while *BASIC–PLUS–2* signals the error "Bad line number pair".
3. You can combine individual line numbers and line ranges in a single DELETE command. Note, however, that a line number range must be followed by a comma and not another hyphen, or BASIC signals an error.

General Rules

1. *BASIC–PLUS–2* signals an error if there are no lines in the specified range. *VAX–11 BASIC* does not signal an error and the DELETE command has no effect.
2. If you do not specify a beginning line number for a range, *VAX–11 BASIC* signals the error "illegal line number". *BASIC–PLUS–2* assumes a beginning line number of 1 and deletes all lines in the range 1 – *lin-num*.
3. If you do not specify an end line number in a range, *VAX–11 BASIC* does not delete any lines and does not signal an error. *BASIC–PLUS–2* deletes only the specified line number.

Examples

```
DELETE 50

DELETE 70-80, 110, 124

DELETE 50,60,90-110
```

9.0 DSKLIB (BASIC–PLUS–2)

Function

The DSKLIB command lets you select a disk-resident, object module library to be used when you build your program. When you use the BUILD command, *BASIC–PLUS–2* includes the specified library in the Task Builder command file. Every system has a disk library default set when *BASIC–PLUS–2* is installed.

Format

```
DSKLIB [ file-spec ]
```

Syntax Rules

1. If you enter the DSKLIB command without a *file-spec*, *BASIC–PLUS–2* prompts for one and displays the name of the current default disk-resident library.

   - *File-spec* can be a disk-resident, object module library supplied with *BASIC–PLUS–2* or a user-created library.
   - If you type a carriage return in response to the prompt, *BASIC–PLUS–2* uses the default disk-resident library.

General Rules

1. The object module libraries supplied by *BASIC–PLUS–2* are LB:BP2OTS.OLB on *RSTS/E systems* and LB:[1,1]BP2OTS.OLB on *RSX–11M/M–PLUS systems*. LB: is a RSTS/E logical name for the library account on disk. These libraries contain the BASIC Object Time System (OTS). OLB is the default object module library file type. If your system does not have memory-resident libraries, the Task Builder extracts all BASIC routines from these disk-resident libraries. See *BASIC on RSX–11M/M–PLUS Systems* and *BASIC on RSTS/E Systems* for more information on object module libraries.

2. The Task Builder links the specified library to your program when you task-build the program. You must use the DSKLIB command before you use the BUILD command to include the library you want in the Task Builder command file.

3. The DSKLIB library you specify is included in all Task Builder command files until you specify a new library with the DSKLIB command or exit from the BASIC environment. When you exit from the BASIC environment, the default object module library set at installation is restored as the default disk-resident library.

4. You can override the DSKLIB command with the /DSKLIB qualifier added to the BUILD command, but the specified library remains in effect for only one BUILD routine.

5. The Task Builder returns an error message when the requested disk-resident library is not available.

6. Consult your system manager for information about the disk-resident libraries available to you.

DSKLIB

Examples

RSX–11M/M–PLUS Systems

```
DSKLIB LB:[1,1]BP2OTS
```

RSTS/E Systems

```
DSKLIB LB:BP2OTS
```

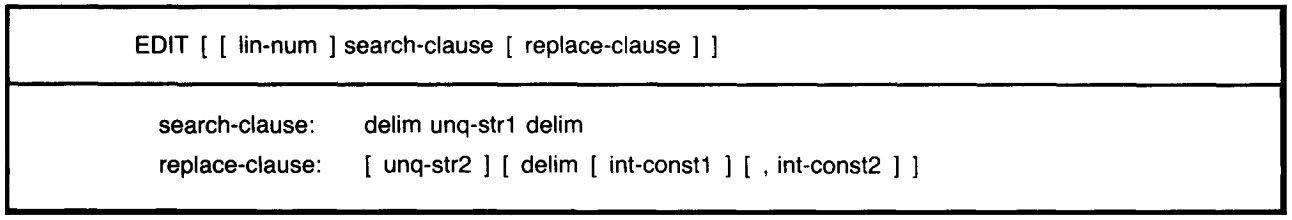
10.0 EDIT

Function

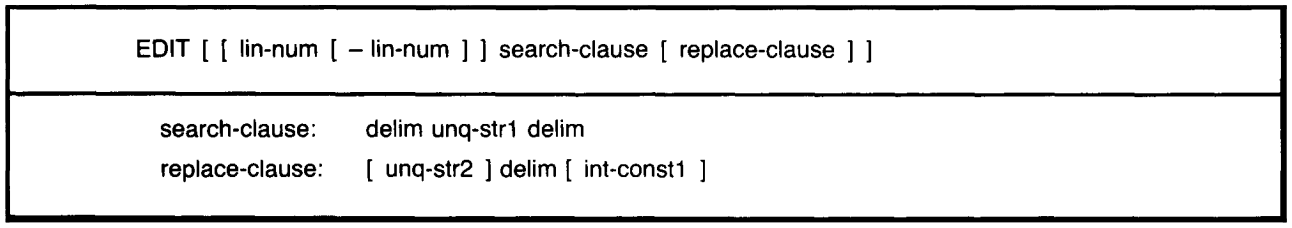
The EDIT command allows you to edit individual program lines in the BASIC environment. In *VAX–11 BASIC*, EDIT with no arguments invokes the default text editor and reads the current program into the editor's buffer. In *BASIC–PLUS–2*, EDIT with no arguments puts you in the *BASIC–PLUS–2* editing mode. *BASIC–PLUS–2* editing mode commands are listed in Table 14 and described in Sections 10.1 to 10.6.

Format

VAX–11 BASIC



BASIC–PLUS–2



Syntax Rules

1. *Lin-num* specifies the line to be edited.
2. *Search-clause* specifies the text you want to remove or replace. *Unq-str1* is the search string you want to remove or replace.
3. *Replace-clause* specifies the replacement text and the occurrence of the search string you want to replace.
   - *Unq-str2* is the replacement string.
   - *Int-const1* specifies the occurrence of *unq-str1* you want to replace. If you do not specify an occurrence, BASIC replaces the first occurrence of *unq-str1*.
4. *Delim* can be any printing character not used in *unq-str1* or *unq-str2*. The examples in this and the following sections use the slash (/) as a delimiter.
5. The *delim* characters in *search-clause* must match, or BASIC signals an error.
6. If the *delim* you use to signal the end of *replace-clause* does not match the *delim* used in *search-clause*, BASIC does not signal an error and treats the end *delim* as part of *unq-str2*.

7. BASIC replaces or removes text in a program line as follows:

   - If *unq-str1* is found, BASIC replaces it with *unq-str2*.
   - If *unq-str1* is not found, BASIC signals an error.
   - If *unq-str1* is null, *VAX–11 BASIC* signals "no change made". *BASIC–PLUS–2* replaces the first character of the last edited line with *unq-str2* and does not signal an error.
   - If *unq-str2* is null, BASIC deletes *unq-str1*. The *delim* in the *replace-clause* is required if you want to delete *unq-str1*.
   - BASIC matches and replaces strings exactly as you type them. If *unq-str1* is uppercase, BASIC searches for an uppercase string. If it is lowercase, BASIC searches for a lowercase string.

VAX *VAX–11 BASIC*

1. The EDIT command followed by a carriage return causes BASIC to temporarily save your program in a file called BASEDITMP.TMP. BASIC then invokes the same editor you use when you type the DCL EDIT command. When you finish editing your program and exit the editor, the edited program is the program currently in memory, and the context of the BASIC environment is unchanged. Note that BASIC deletes all versions of BASEDITMP.TMP when you return to BASIC from the editor.

2. *Int-const2* in *replace-clause* specifies the sub-line of a block of program code where you want BASIC to begin the search.

BP2 *BASIC–PLUS–2*

1. The EDIT command followed by a carriage return puts you in the *BASIC–PLUS–2* editing mode. Editing mode commands, listed in Table 14 and described in Sections 10.1 to 10.6, are valid only in the *BASIC–PLUS–2* editing mode. The editing mode prompt is an asterisk (*).

2. *BASIC–PLUS–2* sets a specified line number as the current edit line, even when the editing operation fails. That line number remains set as the current edit line until you specify another line number or exit the BASIC environment.

3. You can edit a range of lines by separating two line numbers with a hyphen. BASIC signals an error and does not edit the specified range if there are spaces between the hyphen and the line numbers.

4. If you specify a range of lines and an occurrence, BASIC replaces each occurrence of *unq-str1* in each line of the range beginning with the specified occurrence. For example:

```
10 PRINT DISPLAY$, DISPLAY$, DISPLAY$
20 PRINT DISPLAY$, DISPLAY$, DISPLAY$
EDIT 10-20 /DISPLAY$/NEW$/2

10 PRINT DISPLAY$, NEW$, NEW$
20 PRINT DISPLAY$, NEW$, NEW$

 "DISPLAY$" replaced by "NEW$",
4 substitutions
```

General Rules

VAX–11 BASIC

1. *VAX–11 BASIC* displays the edited line with changes after the EDIT command successfully executes.
2. If you specify a *lin-num* with no text parameters, *VAX–11 BASIC* displays the line.

BASIC–PLUS–2

1. *BASIC–PLUS–2* displays the edited line or lines with changes after the EDIT command successfully executes. It also displays a message showing the search string, replacement string, and number of replacements made.
2. If you want to edit a range of numbers, you must specify both the beginning and end of the range. *BASIC–PLUS–2* does not default to the last edited line or to the last line number in the program.
3. When you specify a *lin-num* with no text parameters, *BASIC–PLUS–2* displays the message "Current edit line is x", where x is the specified *lin-num*.
4. When you type EDIT with no parameters to enter the editing mode, *BASIC–PLUS–2* checks the last edited line number to make sure that it still exists in the current program. If it has been deleted, *BASIC–PLUS–2* displays the message "?No current line".

Examples

VAX–11 BASIC

```
EDIT 100 /LEFT$/RIGHT$/3,2

EDIT
```

BASIC–PLUS–2

```
EDIT 300-400 /LEFT$//

EDIT 300 /LEFT$/RIGHT$/3

EDIT
```

(BP2) **Table 14: BASIC–PLUS–2 Editing Mode Commands**

Command	Function
DEFINE	Used to enter a macro definition. A macro definition consists of editing commands. You cannot, though, use the DEFINE and EXECUTE commands in a macro definition. To end the macro definition, type a carriage return and then EXIT or CTRL/Z. You must use the EXECUTE command to execute the macro definition.
EXECUTE	Executes the macro defined by the DEFINE command as many times as you specify.
EXIT (or CTRL/Z)	Allows you to exit from editing mode, execute an INSERT command, or end a DEFINE command.
FIND	Searches from the last edited line to the end of the current program for a specified string.
INSERT	Allows you to add program lines after a specified line number. Type a carriage return and EXIT or CTRL/Z to execute this subcommand.
SUBSTITUTE	Performs the same function and accepts the same text parameters as the EDIT command; you cannot, however, specify line numbers or line number ranges.

10.1 DEFINE (BASIC–PLUS–2)

Function

The DEFINE editing mode command allows you to enter a macro definition. The macro consists of a series of editing mode commands in the order in which they are to execute.

Format

```
{ D      }
{ DEFINE }
```

Syntax Rules

1. The macro definition must consist of valid editing mode commands or *BASIC–PLUS–2* signals an error. You cannot use the DEFINE or EXECUTE editing mode commands in a macro definition.

General Rules

1. Type the DEFINE command and a carriage return, then enter your macro definition. Type EXIT or CTRL/Z in response to the DEFINE prompt (–>) when you have finished entering your macro definition. *BASIC–PLUS–2* displays the editing mode prompt, and you can enter more editing commands.
2. BASIC writes the macro definition to a file, so the definition remains in effect until you enter another DEFINE command. That is, an EXECUTE command executes the last defined macro definition.

Examples

```
*DEFINE

  Enter command sequence:
  ->FIND REM
  ->SUBSTITUTE /REM/!/
  ->EXIT

*
```

10.2 EXECUTE (BASIC–PLUS–2)

Function

The EXECUTE editing mode command executes the last macro defined by the DEFINE command. You specify the number of times the macro is to execute.

Format

```
{ EXE     }
{ EXECUTE } [ int-const ]
```

Syntax Rules

1. *Int-const* specifies the number of times the macro executes. If you do not specify *int-const*, *BASIC–PLUS–2* executes the macro once.

General Rules

1. An EXECUTE command always executes the last defined macro definition. If no macro definition exists, *BASIC–PLUS–2* signals the error "Command sequence has not been defined".

Examples

```
*EXECUTE 5
```

10.3 EXIT or CTRL/Z (BASIC–PLUS–2)

Function

The EXIT or CTRL/Z editing mode command marks the end of a DEFINE or INSERT command or exits from editing mode.

Format

```
{ E    }
{ EXIT }
```

Syntax Rules

None.

General Rules

1. If you type EXIT or CTRL/Z in response to the editing mode prompt, *BASIC–PLUS–2* exits from editing mode.
2. If you type EXIT or CTRL/Z to end a DEFINE or INSERT command, *BASIC–PLUS–2* displays the editing mode prompt and you can enter more editing commands.

Examples

```
*DEFINE

  Enter command sequence
  ->FIND REM
  ->SUBS /REM/!
  ->EXIT

*
```

10.4 FIND (BASIC–PLUS–2)

Function

The FIND editing mode command searches the current program for a specified string starting at the last edited line and continuing to the end of the program.

Format

```
{ F    }
{ FIND }  [ unq-str ]
```

Syntax Rules

1. The FIND command does not require character delimiters for *unq-str*. Delimiters are the space after the command and a carriage return.

General Rules

1. When *unq-str* is found, *BASIC–PLUS–2* displays the line that contains the *unq-str*, sets it as the last edited line, and displays an informational message.
2. If *unq-str* is not found, the last edited line remains unchanged and *BASIC–PLUS–2* displays a message telling you that the string was not found.
3. The FIND command matches *unq-str* exactly as you type it. If *unq-str* is uppercase, *BASIC–PLUS–2* searches for uppercase characters. The delimiters (space and carriage return) are not included in the match.
4. If you do not specify an *unq-str*, the FIND command matches the *unq-str* specified by the last FIND command. If there is no previous FIND command, *BASIC–PLUS–2* matches the first character of the last edited line.

Examples

```
*FIND PRIMT

330     PRIMT 'How many receipts do you have';RECEIPTS

 "PRIMT" found on line 330
*
```

10.5 INSERT (BASIC–PLUS–2)

Function

The INSERT editing mode command allows you to add lines to a program.

Format

```
{ I      }
{ INSERT } [ lin-num ]
```

Syntax Rules

1. *Lin-num* specifies the line number after which you want to insert new program lines. If you do not specify a *lin-num*, BASIC defaults to the last edited line.
2. If *lin-num* does not exist in the source program currently in memory, BASIC signals an error.

General Rules

1. Type in program lines, beginning with a line number, after entering the INSERT command. When you are finished inserting lines, type EXIT or CTRL/Z to return to the editing mode. *BASIC–PLUS–2* displays the editing mode prompt and you can enter more editing mode commands.
2. If you insert a line number that already exists, *BASIC–PLUS–2* replaces the existing line with the code you insert and does not signal a warning.
3. *BASIC–PLUS–2* does not perform syntax checks on inserted program lines even when syntax checking is enabled.
4. The current edit line does not change. For example, if the current edit line is 10 and you insert lines 20 and 30, line 10 remains the current edit line.

Examples

```
*INSERT 30

 Enter lines to be added after line 30
->40 INPUT 'More receipts';RECEIPTS$
->50 IF RECEIPTS$ = ""
->    THEN GOTO 32767
->    END IF
->EXIT
*
```

10.6 SUBSTITUTE (BASIC–PLUS–2)

Function

The SUBSTITUTE editing mode command allows you to substitute one character string for another in the program currently in memory. SUBSTITUTE is the editing mode equivalent of the EDIT command with one exception: you cannot specify a range of lines. The SUBSTITUTE subcommand can replace only one occurrence of the specified search string, while the EDIT command can replace all occurrences in a range of lines, if you so specify.

Format

```
{ S          }
{ SUBSTITUTE } search-clause [ replace-clause ]
```

```
search-clause:     delim unq-str1 delim
replace-clause:    [ unq-str2 ] delim [ int-const ]
```

Syntax Rules

1. *Delim* marks the beginning and end of the search and replace strings. Delimiters are required before and after *unq-str1*. The delimiter after *unq-str2* is optional.

   - *Delim* can be any printing character not used in the search or replace strings. The examples in this section use the slash (/) as a delimiter.
   - The beginning and ending *delim* characters must match, or BASIC signals an error.

2. *Unq-str1* specifies the string you want to remove or replace. *Unq-str2* specifies the string to be substituted for *unq-str1*.

   - If *unq-str1* is found, BASIC replaces it with *unq-str2*.
   - If *unq-str1* is not found, BASIC signals an error.
   - If you do not specify *unq-str2*, BASIC deletes *unq-str1*.
   - If you do not specify *unq-str1*, BASIC replaces the first character of the last edited line with *unq-str2*.
   - The SUBSTITUTE subcommand matches and replaces strings exactly as you type them. If *unq-str1* is uppercase, BASIC searches for an uppercase string. If it is lowercase, BASIC searches for a lowercase string.

3. *Int-const* specifies the occurrence of *str-lit1* you want to replace. If you do not specify an *int-const*, BASIC replaces the first occurrence of *str-lit1*.

4. If you type only the SUBSTITUTE subcommand and a carriage return, *BASIC–PLUS–2* signals the error "Parameters required".

General Rules

1. BASIC displays the edited line with changes after the SUBSTITUTE command executes.

Examples

```
*SUBSTITUTE /A%/ABSOLUTE%/3
```

11.0 EXIT

Function

The EXIT command or CTRL/Z clears memory and returns control to the operating system.

Format

```
EXIT
```

Syntax Rules

None.

General Rules

1. If you type EXIT after creating a new program or editing an old program without first typing SAVE or REPLACE, BASIC signals "Unsaved change has been made, CTRL/Z or EXIT to exit". The message warns you that the new or revised program will be lost if you do not SAVE or REPLACE it. If you type EXIT again, BASIC exits from the environment whether you have saved your changes or not.

Examples

```
EXIT
```

12.0 HELP

Function

The HELP command displays on-line documentation for BASIC commands, keywords, statements, functions, and conventions.

Format

```
HELP [ unq-str ] ...
```

Syntax Rules

1. If you type HELP with no parameters, BASIC displays a list of topics.
2. *Unq-str* is BASIC topic, keyword, command, statement, function, or convention.
3. The first *unq-str* must be a topic. If it is not, BASIC displays a list of topics for you to choose from.
4. You can specify a subtopic after the topic. Separate one *unq-str* from another with a space.
5. You can use the asterisk (*) wildcard character in *unq-str* or alone as *unq-str*. If you use an asterisk in *unq-str*, BASIC displays information on all topics that match the specified portion of *unq-str*. If you use the asterisk alone, BASIC displays information on all BASIC topics.

General Rules

1. If the *unq-str* you specify is not a unique topic or subtopic, BASIC displays a information on all topics or subtopics beginning with *unq-str*. For example:

```
Ready
HELP STATEMENTS CH

STATEMENTS

  CHAIN

     The CHAIN statement transfers control from the current program to another
     BASIC program. The program to which you CHAIN must be in executable
     format.

     Format

     CHAIN <str-exp>

     Example

     240 CHAIN "COSINE.EXE"
```

(continued on next page)

```
STATEMENTS

  CHANGE

     The CHANGE statement: 1) converts a string of characters to their ASCII
     integer values,  or 2) converts a list of numbers to a string of ASCII
     characters.

     Format

     String Variable to Array:

         CHANGE str-exp TO num-array

     Array to String Variable:

         CHANGE num-array TO str-vbl

     Example

     200 CHANGE ARRAY_CHANGES TO A$

Topic?
```

2. An asterisk (*) indicates that you want to display information that matches any portion of the topic you specify. For example, if you type HELP GO*, BASIC displays information on GOSUB and GOTO.
3. When information on a particular topic or subtopic is not available, BASIC signals the message "Sorry, no documentation on *unq-str*" and a list of "Additional information available".

Examples

```
HELP STATEMENTS ON GOTO

STATEMENTS

  ON

    GOTO

      The ON GOTO statement transfers program control to one of several lines,
      depending on the value of a control expression.

        Format

                        { GO TO }
            ON int-exp  { GOTO  } target ,... [ OTHERWISE target ]

        Example

        330 ON INDEX% GOTO 700,800,900, OTHERWISE 1000

Topic?
```

13.0 IDENTIFY

Function

The IDENTIFY command displays an identification header on the controlling terminal. The header contains the name and version number of BASIC.

Format

```
IDENTIFY
```

General Rules

1. The message displayed by the IDENTIFY command includes the name of the BASIC compiler and the version number.

Examples

VAX–11 BASIC

```
IDENTIFY

VAX-11 BASIC V2.0
```

BASIC–PLUS–2

```
IDENTIFY

PDP-11 BASIC-PLUS-2 V2.0
```

14.0 INQUIRE

Function

The INQUIRE command is a synonym for the HELP command. See the HELP command for syntax rules.

15.0 LIBRARY (BASIC–PLUS–2)

Function

The LIBRARY command allows you to specify a memory-resident *BASIC–PLUS–2* or user-created library to be used when you task-build the program. When you use the BUILD command, *BASIC–PLUS–2* includes the specified library in the Task Builder command file. The default library for the LIBRARY command is chosen by your system manager when *BASIC–PLUS–2* is installed.

Format

```
LIBRARY [ lib-param ]

lib-param:  { file-spec }
            { lib-nam  }
            { NONE     }
```

Syntax Rules

1. If you enter the LIBRARY command without a *lib-param*, *BASIC–PLUS–2* prompts for one and displays the name of the current default memory-resident library.

   - *Lib-nam* or *file-spec* can be a memory-resident library supplied by *BASIC–PLUS–2* or a user-created library. If you specify only *lib-nam* with no device, *BASIC–PLUS–2* assumes LB: on *RSTS/E systems* and LB:[1,1] on *RSX–11M/M–PLUS systems*.

   - NONE tells the Task Builder not to link your task to the BASIC default memory-resident library. Therefore, the Task Builder links to the BASIC disk-resident, object module library, BP2OTS.OLB.

   - If you type a carriage return in response to the prompt, the current default memory-resident library is used.

General Rules

1. The memory-resident libraries supplied by *BASIC–PLUS–2* are BP2RES and BP2SML. Because memory-resident libraries are optional, your system manager can select none, one, or both then *BASIC–PLUS–2* is installed. See *BASIC on RSX–11M/MPLUS Systems* or *BASIC on RSTS/E Systems* for information on using *BASIC–PLUS–2* memory-resident libraries. See your system manager for information on the libraries available on your system.

2. On *RSTS/E* systems, the LIBRARY command does not require the LB: logical name. BASIC automatically searches this account for the memory-resident library symbol table. On *RSX–11M/M–PLUS systems*, the LIBRARY command automatically references libraries on LB:[1,1] unless you specify another UIC.

3. *BASIC–PLUS–2* links the specified library to your program when you task-build the program. You must use the LIBRARY command before you use the BUILD command to include the specified library in the Task Builder command file.

4. The library you specify is included in your Task Builder command files until you specify a new library with the LIBRARY command or exit from the compiler. When you exit from the compiler, the original default library is restored as the default.
5. You can override the LIBRARY command with the /LIBRARY qualifier added to the BUILD command, but the specified library remains in effect for only one BUILD routine.
6. The Task Builder returns an error message when the requested resident library is not available.

Examples

```
LIBRARY BP2RES
```

16.0 LIST and LISTNH

Function

The LIST command displays the program lines of the program currently in memory. Line numbers are sequenced in ascending order. LISTNH displays program lines without the program header.

Format

VAX-11 BASIC

```
{ LISTNH }
{ LIST   }  [ lin-num [ sep [ lin-num ] ] ] ...

sep:  { - }
      { , }
```

VAX

BASIC-PLUS-2

```
{ LISTNH }
{ LIST   }  [ [ - ] lin-num ] [ sep [ lin-num ] ]...

sep:  { - }
      { , }
```

BP2

Syntax Rules

1. The LIST command displays program lines, along with a header containing the program name, the current time, and the date. To suppress the program header, type LISTNH.
2. LIST without parameters displays the entire program.
3. The *sep* characters allow you to display single lines or a range of lines.
   - To display single lines, separate line numbers with commas. For example:

     ```
     LIST 30,70
     ```

     displays a header and lines 30 and 70.
   - To display an inclusive range of lines, separate line numbers with a hyphen. The first number must be lower than the second number in the range or BASIC signals an error. For example:

     ```
     LIST 30-70
     ```

     displays lines 30 through 70.
4. Line number ranges must be separated from other ranges or individual line numbers by commas as BASIC does not allow two consecutive hyphens.

VAX *VAX–11 BASIC*

1. A *lin-num* followed by a hyphen and a carriage return displays the specified line and all remaining lines in the program.
2. A hyphen between the LIST command and *lin-num* causes *VAX–11 BASIC* to signal an error.

BP2 *BASIC–PLUS–2*

1. A hyphen between the LIST command and the *lin-num* displays all lines from the beginning of the program up to and including the *lin-num* you specify.
2. A *lin-num* followed by a comma or a hyphen and a carriage return displays only the specified line.
3. If there are no lines in the specified range, *BASIC–PLUS–2* signals an error.

General Rules

1. BASIC displays the source program lines in the order you specify in the command line. That is, BASIC displays line 100 before line 10 if you type LIST 100,10.

Examples

VAX–11 BASIC

```
LIST 50, 200-300, 30000-
```

BASIC–PLUS–2

```
LISTNH -30, 2000-2500, 19000
```

17.0 LOAD

Function

The LOAD command makes a previously created object module or modules available for execution with the RUN command.

Format

```
LOAD file-spec [ + file-spec ] ...
```

Syntax Rules

1. *File-spec* must be a BASIC object module or BASIC signals an error. OBJ is the default file type. If you specify only the file name, BASIC searches for an OBJ file in the current default directory.
2. Each device and directory specification applies to all following file specifications until you specify a new directory or device.
3. Each new LOAD command cancels the effect of a previous LOAD command. That is, the LOAD command clears all previously loaded object modules from memory.
4. The LOAD command accepts multiple device, directory, and file specifications.

General Rules

1. BASIC does not process the loaded object files until you issue the RUN command. Consequently, errors in the loaded modules may not be detected until you execute them.
2. BASIC signals an error:
   - If the file is not found
   - If the file specification is not valid
   - If the file is not a BASIC object module
   - If run-time memory is exceeded

   Errors do not change the program currently in memory.
3. Typing the LOAD command does not change the program currently in memory.

Examples

```
LOAD PROGA + PROGB + PROGC
```

18.0 LOCK

Function

The LOCK command changes default values for COMPILE command qualifiers. It is a synonym for the SET command. See the SET command for syntax rules.

19.0 NEW

Function

The NEW command clears BASIC memory and allows you to assign a name to a new program.

Format

```
NEW [ prog-nam ]
```

Syntax Rules

1. *Prog-nam* is the name of the program you want to create. *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* allow names to contain up to nine alphanumeric characters. *BASIC–PLUS–2* on *RSTS/E systems* allows names to contain up to six alphanumeric characters.
2. *BASIC–PLUS–2 on RSTS/E systems* truncates a *prog-nam* that exceeds six characters and does not signal an error.
3. *VAX–11 BASIC* and *BASIC–PLUS–2 on RSX–11M/M–PLUS systems* signal an error if the *prog-nam* exceeds nine characters.
4. *VAX–11 BASIC* signals "error in program name" if you specify a file type. *BASIC–PLUS–2* ignores the file type and does not signal an error.

General Rules

1. If you do not specify a *prog-nam*, BASIC prompts with:

   ```
   New file name--
   ```

2. The default name is NONAME. If you do not provide a *prog-nam* in response to the prompt, BASIC assigns the file name NONAME to your program.
3. When you type the NEW command, the program currently in memory is lost. Program modules loaded with the LOAD command remain unchanged.

Examples

```
NEW PROG1
```

20.0 ODLRMS (BASIC–PLUS–2)

Function

The ODLRMS command allows you to select an overlay description (ODL) file to describe the RMS overlay structure to be used when your program is task built. When you use the BUILD command, *BASIC–PLUS–2* includes the specified ODL file in the Task Builder command file. Every system has an ODL default set when *BASIC–PLUS–2* is installed. See your system manager for the name of your BASIC default.

Format

```
ODLRMS [ odl-param ]

odl-param:   { file-spec }
             { NONE      }
```

Syntax Rules

1. If you enter the ODLRMS command without an *odl-param*, *BASIC–PLUS–2* prompts for one and displays the name of the current default ODL file.

   - *File-spec* can be an ODL file supplied by RMS or a user-created file. Table 15 lists and describes RMS ODL files.
   - NONE tells the Task Builder not to link your task to any RMS ODL file.
   - If you type a carriage return in response to the prompt, *BASIC–PLUS–2* uses the default ODL file.

General Rules

1. New versions of RMS can change ODL file names, so consult the RMS distribution kit for current ODL names. LB: is a RSTS/E logical name for the library account on disk. On *RSX–11M/M–PLUS systems*, you must specify LB:[1,1] before the ODL file name.
2. Enter the ODLRMS command before you enter the BUILD command. The ODL file you specify is included in all Task Builder command files until you enter a new ODLRMS command or exit from the BASIC environment, at which time *BASIC–PLUS–2* returns to the ODL default file.
3. You can override the ODLRMS command with the ODL qualifier to the BUILD command for a single BUILD operation.
4. Refer to the RMSRES compiler command to see which ODL files are required for each RMS library.

5. The Task Builder returns an error message if the ODL file you select is not available or valid. Consult your system manager for information about ODL files available to you.

6. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on using RMS libraries.

Table 15: ODL Files

ODLRMS Option	File Organization			Type of Library	Overlay Segments
	Seq	Rel	Ind		
RMSRLX	Yes	Yes	Yes	Memory	None
DAPRLX	Yes	Yes	Yes	Memory	None
RMS11S	Yes	Yes	No	Disk	11
RMS12S	Yes	Yes	No	Disk	5
RMS11X	Yes	Yes	Yes	Disk	35
RMS12X	Yes	Yes	Yes	Disk	13
DAP11X	Yes	Yes	Yes	Disk	16

Examples

RSX–11M/M–PLUS Systems

```
ODLRMS LB:[1,1]RMSRLX.ODL
```

RSTS/E Systems

```
ODLRMS LB:RMSRLX.ODL
```

21.0 OLD

Function

The OLD command brings a previously created BASIC program into memory.

Format

```
OLD [ file-spec ]
```

Syntax Rules

1. If you do not name a *file-spec*, BASIC prompts for one. If you do not enter a *file-spec* in response to the prompt, BASIC searches for a file named NONAME.BAS (*VAX–11 BASIC*) or NONAME.B2S (*BASIC–PLUS–2*) in the current default directory.
2. The default file type is BAS for *VAX–11 BASIC* and B2S for *BASIC–PLUS–2*.

General Rules

1. If the compiler cannot find the *file-spec*, *VAX–11 BASIC* signals the error "file not found" and *BASIC–PLUS–2* signals "can't find file or account".
2. When the specified file is found, it is placed in memory and any program currently in memory is erased. If BASIC does not find the specified file, the program currently in memory does not change.
3. If you specify a file that does not begin with a line number, BASIC discards all text up to the first line number, brings the file into memory, and signals the error "Non-continued statement has no line number near <line number>". You can then LIST and SAVE the program.

Examples

```
OLD CHECK
Ready
```

22.0 Qualifiers

Function

BASIC qualifiers allow you to specify defaults for the compilation process and the BASIC environment. You specify qualifiers with the COMPILE and SET commands. In *BASIC–PLUS–2*, you can also specify qualifiers with the BUILD and RUN commands.

Format

```
command [ /qualifier ] ...
```

Syntax Rules

1. The slash delimiter is not required before the first qualifier in the SET command. Multiple qualifiers, however, must be separated by slashes or commas. See the syntax rules for the SET command for more information on separating qualifiers.
2. You can abbreviate all positive qualifiers to the first three letters of the qualifier keyword. You can abbreviate a negative qualifier to NO and the first three letters of the qualifier keyword.

General Rules

1. Table 16 lists *VAX–11 BASIC* qualifiers and their functions. Table 17 lists *BASIC–PLUS–2* qualifiers, the commands they can be used with, and their functions.
2. In cases of ambiguous or erroneous qualifiers, *VAX–11 BASIC* signals the error "Unknown qualifier", while *BASIC–PLUS–2* signals "Illegal switch".
3. When you exit from the BASIC environment, all defaults set with qualifiers return to the defaults. Use the SHOW command before setting any qualifiers to display your system defaults.

Examples

```
COMPILE/NOOBJ/DOUBLE/DEBUG

SET /TYPE_DEFAULT: EXPLICIT/LIST
```

Table 16: VAX–11 BASIC COMPILE and SET Command Qualifiers

Qualifier	Function
[NO]ANSI_STANDARD	Tells BASIC to compile programs according to the ANSI Minimal BASIC Standard and to flag syntax that does not conform to the standard. See *BASIC on VAX/VMS Systems* for information on the ANSI Minimal BASIC Standard.
[NO]AUDIT [sep text-entry] sep: { : = } text-entry: { str-lit file-spec }	Tells BASIC to include a history entry in the CDD data base when a CDD definition is extracted. *Str-lit* is a quoted string. *File-spec* is a text file. The history entry includes: • The contents of *str-lit*, or up to the first 64 lines in the file specified by *file-spec* • The name of the program module, process, user name, and user UIC that accessed the CDD • The time and date of the access • A note that access was made by the BASIC compiler • A note that the access was an extraction
[NO]BOUNDS_CHECK	Tells BASIC to perform range checks on array subscripts. BASIC checks that all subscript references are within the array boundaries set when the array was declared.
BYTE	Causes the compiler to allocate eight bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as BYTE values and must be in the BYTE range or BASIC signals the error "Integer error or overflow". Table 2 in this manual lists BASIC data types and ranges.
[NO]CROSS_REFERENCE [sep [NO]KEYWORDS] sep: { : = }	Causes the compiler to include cross-reference information in the program listing file. If you specify KEYWORDS, BASIC also cross-references BASIC keywords used in the program. The listing file takes the program name as the file name and a default file type of LIS.
[NO]DEBUG	Tells BASIC to provide records for the *VAX–11 Symbolic Debugger*. See *BASIC on VAX/VMS Systems* for information on using the *VAX–11 Symbolic Debugger*.
DECIMAL_SIZE sep (d,s) sep: { : = }	Allows you to specify the default size and precision for all DECIMAL data not explicitly assigned size and precision in the program. You specify the total number of digits (d) and the number of digits to the right of the decimal point (s). BASIC signals the error "Decimal error or overflow" (ERR=181) when DECIMAL values are outside the range specified with this qualifier. See Table 2 in this manual and Appendix C in *BASIC on VAX/VMS Systems* for information on the storage and range of packed decimal data.

Table 16: VAX–11 BASIC COMPILE and SET Command Qualifiers (Cont.)

Qualifier	Function
DOUBLE	Causes the compiler to allocate 64 bits of storage in D_FLOAT format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as DOUBLE values and must be in the DOUBLE range or BASIC signals the error "Floating-point error or overflow". Table 2 in this manual lists BASIC data types and ranges.
[NO]FLAG [sep (flag-clause,...)] sep: { : | = } flag-clause: { [NO]BP2COMPATIBILITY | [NO]DECLINING }	Causes BASIC to provide compile-time information about program elements that are not compatible with *BASIC–PLUS–2* or that DIGITAL designates as declining. An element is designated declining when BASIC has a preferred and often more powerful way to perform the operation. If you specify BP2COMPATIBILITY, BASIC will flag the following source code as incompatible with *BASIC–PLUS–2*: • String comparisons using (<),(>),(<=), or (>=) • DECIMAL keyword and DECIMAL function • HFLOAT keyword • GFLOAT keyword • LOC function • MAR and MAR% functions • MARGIN and NOMARGIN statements • RECORD declarations • More than 16 digits of precision in a floating-point literal • Explicit literal notation that specifies a radix • Explicit literal notation with data type other than "B" (BYTE), "W" (WORD), "L" (LONG), "S" (SINGLE), "D" (DOUBLE), or "C" (CHARACTER) • Names in the EXTERNAL statement that have more than six characters or contain characters not in the Radix-50 character set • BY DESC clauses on anything other than entire arrays or unsubscripted STRING variables • BY VALUE clauses for anything other than BYTE or WORD unsubscripted variables • More than eight parameters to a DEF, subprogram, or external function

(continued on next page)

Table 16: VAX–11 BASIC COMPILE and SET Command Qualifiers (Cont.)

Qualifier	Function
	• Arrays of more than eight dimensions • Terminal-format files opened with no MAP or RECORDSIZE clause and no ACCESS READ clause • BY DESC, BY REF, and BY VALUE clauses in SUB and FUNCTION statements If you specify DECLINING, BASIC will flag the following source code as declining: • CVT$$ (use EDIT$) • CVT$%, CVT$F, CVT%$, CVTF$, and SWAP% (use multiple MAP statements) • DEF* functions (use DEF functions) • FIELD statements (use MAP DYNAMIC and REMAP) • GOTO lin-num% (do not use the integer suffix with a line number)
GFLOAT	Causes the compiler to allocate 64 bits of storage in G_FLOAT format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as GFLOAT values and must be in the GFLOAT range or BASIC signals "Floating-point error or overflow". Table 2 in this manual lists BASIC data types and ranges.
HFLOAT	Causes the compiler to allocate 128 bits of storage in H_FLOAT format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as HFLOAT values and must be in the HFLOAT range or BASIC signals "Floating-point error or overflow". Table 2 in this manual lists BASIC data types and ranges.
[NO]LINE	Includes line number information in object modules. If you specify NOLINE in a program containing a RESUME statement or using the run-time ERL function, BASIC warns that the NOLINE qualifier has been overridden.
[NO]LIST	Tells BASIC to produce a source listing file. By default, this file contains a memory allocation map. The listing file takes the name of the program and a default file type of LIS.

(continued on next page)

Table 16: VAX–11 BASIC COMPILE and SET Command Qualifiers (Cont.)

Qualifier	Function
LONG	Causes the compiler to allocate 32 bits of storage as the default size for all integer data not explicitly typed in the program. Untyped integer values are treated as LONG values and must be in the LONG range or BASIC signals the error "Integer error or overflow". Table 2 in this manual lists BASIC data types and ranges.
[NO]MACHINE_CODE	Causes BASIC to include the machine code generated by the compilation in the program listing file.
[NO]OBJECT	Generates an object module with the same file name as the program and a default file type of OBJ. Use NOOBJECT to check your program for errors without creating an object file.
[NO]OVERFLOW [sep (data-type,...)] sep: { : | = } data-type: { INTEGER | DECIMAL }	Tells BASIC to report arithmetic overflow for operations on integer and/or packed decimal data.
[NO]ROUND	Tells BASIC to round rather than truncate DECIMAL values.
[NO]SETUP	Tells BASIC to make calls to the Run-Time Library that set up the stack for BASIC variables, set up dynamic string and array descriptors, initialize variables, and enable BASIC error handling. If you specify NOSETUP, BASIC will attempt to optimize your program by omitting these calls. If your program contains any of the following elements, BASIC provides an informational diagnostic and does not optimize your program: • CHANGE statements • DEF or DEF* statements • Dynamic string variables • Executable DIM statements • EXTERNAL string functions • MAT statements • MOVE statements for an entire array • ON ERROR statements

(continued on next page)

Table 16: VAX–11 BASIC COMPILE and SET Command Qualifiers (Cont.)

Qualifier	Function
	• READ statements • REMAP statements • RESUME statements • String concatenation • Built-in string functions • Virtual array declarations Note that program modules compiled with NOSETUP cannot perform any I/O and have no error handling capabilities. If an error occurs in such a module, the error is resignaled to the calling program.
[NO]SHOW [sep (show-item,...)] sep: { : | = } show-item: { [NO]CDD_DEFINITIONS | [NO]ENVIRONMENT | [NO]INCLUDE | [NO]MAP | [NO]OVERRIDE }	Tells BASIC what to include in the listing file: • CDD_DEFINITIONS specifies translated CDD definitions. • ENVIRONMENT specifies a listing of the compilation qualifiers in effect. • INCLUDE specifies a listing of the contents of %INCLUDE files. • MAP specifies a storage allocation map. • OVERRIDE cancels the effect of all %NOLIST directives in the source program. If you do not specify a *show-item*, BASIC uses the defaults set with the DCL command.
SINGLE	Causes the compiler to allocate 32 bits of storage in F_FLOAT format as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as SINGLE values and must be in the SINGLE range or BASIC signals the error "Floating-point error or overflow". Table 2 in this manual lists BASIC data types and ranges.
[NO]SYNTAX_CHECK	Tells BASIC to perform syntax checking after each program line is typed.
[NO]TRACEBACK	Causes BASIC to include traceback information in the object file that allows reporting of the sequence of calls that transferred control to the statement where an error occured.

(continued on next page)

Table 16: VAX–11 BASIC COMPILE and SET Command Qualifiers (Cont.)

Qualifier	Function
TYPE_DEFAULT sep default-clause sep: { : | = } default-clause: { REAL | INTEGER | DECIMAL | EXPLICIT }	Sets the default data type (REAL, INTEGER, or DECIMAL) for all data not explicitly typed in your program or specifies that all data must be explicitly typed (EXPLICIT). • REAL specifies that all data not explicitly typed is floating-point data of the default size (SINGLE, DOUBLE, GFLOAT, or HFLOAT). • INTEGER specifies that all data not explicitly typed is integer data of the default size (BYTE, WORD, or LONG). • DECIMAL specifies that all data not explicitly typed is packed decimal data of the default size. • EXPLICIT specifies that all data in a program must be explicitly typed. Implicitly declared variables cause BASIC to signal an error.
VARIANT sep int-const sep: { : | = }	Establishes *int-const* as a value to be used in compiler directives. The variant value can be referenced in a lexical expression by using the lexical function, %VARIANT. *Int-const* always has a data type of LONG.
[NO]WARNINGS [sep warn-clause] sep: { : | = } warn-clause: { [NO]WARNINGS | [NO]INFORMATIONALS }	Tells BASIC to display warning and/or informational messages. If you specify WARNINGS but do not specify a *warn-clause*, BASIC displays both warnings and informational messages.
WORD	Causes the compiler to allocate 16 bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as WORD values and must be in the range –32768 to 32767 or BASIC signals the error "Integer error or overflow." Table 2 in this manual lists BASIC data types and ranges.

Table 17: BASIC–PLUS–2 Command Qualifiers

Qualifier	Commands	Function
BRLRES: lib-param lib-param: {file-spec / NONE}	BUILD	Lets you specify a memory-resident library to be linked to your program. *File-spec* can be a library supplied with *BASIC–PLUS–2* or a user-created library. NONE tells the Task Builder not to link your task to the default memory-resident library. See the BRLRES command syntax rules in this manual for more information on memory-resident libraries.
BYTE	COMPILE RUN SET	Causes the compiler to allocate eight bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as BYTE values and must be in the BYTE range or BASIC signals the error "Integer error or overflow". Table 2 in this manual lists BASIC data types and ranges.
[NO]CHAIN	COMPILE** RUN** SET**	Enables other programs to CHAIN into the program using the LINE clause of the CHAIN statement. The default (CHAIN or NOCHAIN) is an installation option. If the program has more than 200 line numbers, NOCHAIN reduces the memory needs of the output program by disabling storage of line numbers in memory. You cannot chain from one DECNET node to another.
[NO]CLUSTER[:lib-param] lib-param: {file-spec / NONE}	BUILD SET	Tells the Task Builder to cluster memory-resident libraries to increase the space available for your task. For the cluster qualifier to have an effect, at least two resident libraries must be linked to the task: the *BASIC–PLUS–2* resident library, and one other resident library. *File-spec* specifies a memory-resident library to be clustered. NONE specifies that only the *BASIC–PLUS–2* and *RMS* libraries are clustered. If there is no default CLUSTER library, the CLUSTER qualifier without a parameter acts the same as the CLUSTER:NONE qualifier. The specified library must be in the account LB: on *RSTS/E systems* or the account LB:[1,1] on *RSX systems*. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on using RMS libraries.
[NO]CROSS_REFERENCE[:[NO]KEYWORDS]	COMPILE SET	Causes the compiler to include cross-reference information in the program listing file. If you specify KEYWORDS, BASIC also cross-references BASIC keywords used in the program. The listing file takes the program name as the file name and a default file type of LST.

(continued on next page)

Table 17: BASIC–PLUS–2 Command Qualifiers (Cont.)

Qualifier	Commands	Function
[NO]DEBUG	COMPILE RUN SET	Tells BASIC to provide records for the *BASIC–PLUS–2* debugger when you compile a program or to pass control to the debugger when you execute a program with RUN in the BASIC environment. The LINE qualifier must be in effect when you compile a program with the DEBUG qualifier in effect.
DOUBLE	COMPILE RUN SET	Causes the compiler to allocate 64 bits of storage as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as DOUBLE values and must be in the DOUBLE range or BASIC signals the error "Floating-point error". Table 2 in this manual lists BASIC data types and ranges.
DSKLIB : file-spec	BUILD	Lets you specify a disk-resident object module library to be linked to your program. *File-spec* can be a library supplied with *BASIC–PLUS–2* or a user-created library. NONE tells the Task Builder not to link your task to the default object module library. See the DSKLIB command syntax rules for more information on disk-resident libraries.
[NO]DUMP	BUILD SET	Tells the Task Builder to generate a memory dump if the program aborts with a fatal error.
EXTEND: int-const	BUILD SET	Specifies the amount of space to be added to the initial task size when the task is started. The Task Builder rounds the extension up to the nearest 32–word boundary. The maximum extension is 32000.
FLAG:[NO]DECLINING	COMPILE RUN SET	Causes BASIC to provide compile-time information about program elements that DIGITAL designates declining. An element is designated declining when BASIC has a preferred and often more powerful capablity. When you specify FLAG:DECLINING, BASIC will flag the following source code: • CVT$$ (use EDIT$) • CVT$%, CVT$F, CVT%$, CVTF$, and SWAP% (use multiple MAP statements) • DEF* functions (use DEF functions)

(continued on next page)

Table 17: BASIC–PLUS–2 Command Qualifiers (Cont.)

Qualifier	Commands	Function
		• FIELD statements (use MAP DYNAMIC and REMAP) • GOTO lin-num% (do not use the integer suffix with a line number)
[NO]IND	BUILD SET	Causes the Task Builder to include the code needed for indexed file operations. *BASIC–PLUS–2* enables this qualifier automatically for programs containing an OPEN statement with an ORGANIZATION INDEXED clause.
LIBRARY: lib-param lib-param: { lib-nam / file-spec / NONE }	BUILD	Lets you specify a memory-resident library to be linked to your program. *File-spec* and *lib-nam* can be a library supplied with *BASIC–PLUS–2* or a user-created library. If you specify only a *lib-nam* with no device, BASIC assumes LB: on *RSTS/E systems* and LB:[1,1] on *RSX systems*. NONE tells the Task Builder not to link your task to the default memory-resident library. Therefore, the Task Builder links to the BASIC disk-resident, object module library, BP2OTS.OLB. See the LIBRARY command syntax rules for more information on memory-resident libraries.
[NO]LINE	COMPILE RUN SET	Includes line number information in object modules. If you specify NOLINE in a program containing a RESUME statement or using the run-time ERL function, BASIC warns that the NOLINE qualifier has been overridden.
[NO]LIST	COMPILE SET	Tells BASIC to produce a source listing file. The listing file takes the name of the program and a default file type of LST.
LONG	COMPILE RUN SET	Causes the compiler to allocate 32 bits of storage as the default size for all integer data not explicitly typed in the program. Untyped integer values are treated as LONG values and must be in the LONG range or BASIC signals the error "Integer error". Table 2 in this manual lists BASIC data types and ranges.
[NO]MACRO	COMPILE SET	Converts the program into MACRO source code and saves it in a file with the same name as the program and a file type of MAC. The MAC file can be assembled.

(continued on next page)

Table 17: BASIC–PLUS–2 Command Qualifiers (Cont.)

Qualifier	Commands	Function
[NO]MAP	BUILD	Includes information for the allocation map in the Task Builder command file.
[NO]OBJECT	COMPILE SET	Generates an object module with the same file name as the program and a default file type of OBJ. Use NOOBJECT to check your program for errors without creating an object file.
ODLRMS: odl-param odl-param: {file-spec | NONE}	BUILD	Lets you specify an ODL file to describe the RMS overlay structure to be used by the Task Builder. *File-spec* can be an ODL file supplied by RMS or a user-created file. NONE tells the Task Builder not to link your task to the default ODL file. See the ODLRMS command syntax rules in this manual for more information on ODL files.
PAGE_SIZE: int-const	COMPILE SET	Sets the page size for the listing file. *Int-const* must be greater than zero or BASIC signals the warning "Listing length out of range — ignored".
[NO]REL	BUILD SET	Causes the Task Builder to include the code needed for relative file operations. *BASIC–PLUS–2* sets this qualifier automatically for programs containing an ORGANIZATION RELATIVE clause in an OPEN statement.
RMSRES: lib-param lib-param: {file-spec | NONE}	BUILD	Lets you specify an RMS library that supplies code for file and record operations to be linked to your program. *File-spec* can be a library supplied by RMS or a user-created library. NONE tells the Task Builder not to link your task to the default RMS library. See the RMSRES command syntax rules for more information on RMS libraries.
[NO]SCALE: const	COMPILE	Allows control of accumulated round-off errors when double precision numbers (values typed DOUBLE) are used. Numbers are stored as multiples of 10 by setting the scale factor (*const*) from 0 to 6. Floating-point numbers are truncated to an integer value of 0 to 6. A scale factor larger than 6 causes BASIC to signal the error message "Scale factor of *n* is out of range."
[NO]SEQ	BUILD SET	Causes the Task Builder to include the RMS–11 code needed for sequential file operations. *BASIC–PLUS–2* sets this qualifier automatically for programs containing an ORGANIZATION SEQUENTIAL clause in the OPEN statement.

(continued on next page)

Table 17: BASIC–PLUS–2 Command Qualifiers (Cont.)

Qualifier	Commands	Function
SINGLE	COMPILE RUN SET	Causes the compiler to allocate 32 bits of storage as the default size for all floating-point data not explicitly typed in the program. Untyped floating-point values are treated as SINGLE values and must be in the SINGLE range or BASIC signals the error "Floating-point error". Table 2 in this manual lists BASIC data types and ranges.
[NO]SYNTAX_CHECK	COMPILE RUN SET	Tells BASIC to perform syntax checking after each program line is typed.
TYPE_DEFAULT: default-clause default-clause: { REAL / INTEGER / EXPLICIT }	COMPILE RUN SET	Sets the default data type (REAL or INTEGER) for all data not explicitly typed in your program or specifies that all data must be explicitly typed (EXPLICIT). • REAL specifies that all data not explicitly typed is floating-point data of the default size (SINGLE or DOUBLE). • INTEGER specifies that all data not explicitly typed is integer data of the default size (BYTE, WORD, or LONG). • EXPLICIT specifies that all data in a program must be explicitly typed. Implicitly declared variables cause BASIC to signal an error.
VARIANT: int-const	COMPILE RUN SET	Establishes *int-const* as a value to be used in compiler directives. The variant value can be referenced i. a lexical expression by using the lexical function, %VARIANT. *Int-const* always has a data type of WORD.
[NO]VIR	BUILD* SET*	Causes the Task Builder to include the RMS code needed for virtual array and block I/O file operations. *BASIC–PLUS–2* sets this qualifier automatically when you compile a program containing an ORGANIZATION VIRTUAL clause in the OPEN statement.

(continued on next page)

Table 17: BASIC–PLUS–2 Command Qualifiers (Cont.)

Qualifier	Commands	Function
WIDTH: int-const	COMPILE SET	Sets the width of the listing file. *Int-const* must be in the range 72 to 132, inclusive, or BASIC signals the warning "Listing width out of range — ignored".
WORD	COMPILE RUN SET	Causes the compiler to allocate 16 bits of storage as the default for all integer data not explicitly typed in the program. Untyped integer values are treated as WORD values and must be in the range –32768 to 32767 or BASIC signals the error "Integer error." Table 2 in this manual lists BASIC data types and ranges.

* *RSX only*
** *RSTS/E only*

23.0 RENAME

Function

The RENAME command allows you to assign a new name to the program currently in memory. BASIC does not write the renamed program to a file until you save the program with the REPLACE or SAVE command.

Format

```
RENAME [ prog-nam ]
```

Syntax Rules

1. *Prog-nam* specifies the new program name. *VAX–11 BASIC* and *BASIC–PLUS–2 on RSX–11M/M–PLUS systems* allow names to contain up to nine alphanumeric characters. *BASIC–PLUS–2 on RSTS/E systems* allows names to contain up to six alphanumeric characters.
2. The program you want to rename must be in memory. If you type RENAME with no program in memory, BASIC renames the default program, NONAME, to the specified *prog-nam*.

VAX–11 BASIC 

1. If you do not specify a *prog-nam*, *VAX–11 BASIC* renames the program currently in memory NONAME.
2. If you specify a file type, *VAX–11 BASIC* signals the error "error in program name".

BASIC–PLUS–2 

1. *BASIC–PLUS–2* prompts for the new *prog-nam* if you do not specify one with the RENAME command. If you do not specify a *prog-nam* in response to the prompt, the name of the program currently in memory remains unchanged.
2. If you specify a file type, *BASIC–PLUS–2* ignores the file type, does not signal an error, and assigns the B2S file type to the file when you save it.

General Rules

1. You must type SAVE or REPLACE to write the renamed program to a file. If you do not type SAVE or REPLACE, BASIC does not save the renamed program.
2. The RENAME command does not affect the original saved version of the program. For example:

```
OLD TEST
Ready

RENAME NEWTES
Ready

SAVE
```

RENAME

In this example, the OLD command calls the program named TEST into memory. The RENAME command renames TEST to NEWTES and the SAVE command writes NEWTES.BAS (*VAX–11 BASIC*) or NEWTES.B2S (*BASIC–PLUS–2*) to a file. The original file, TEST.BAS or TEST.B2S, is not changed and is not deleted from your account.

Examples

```
RENAME NEWPRO
```

24.0 REPLACE

Function

The REPLACE command writes the current program to a storage medium.

Format

```
REPLACE [ file-spec ]
```

Syntax Rules

1. If you do not supply a *file-spec*, BASIC writes the program to the default disk with the file name of the program currently in memory.

   - *VAX–11 BASIC* and *BASIC–PLUS–2 on RSX–11M/M–PLUS systems* create and save a new version of the file, incrementing the version number by one. Previous versions of the file remain unchanged.
   - *BASIC–PLUS–2 on RSTS/E systems* overwrites the original version of the file with the new version.

General Rules

1. The *file-spec* does not have to match that of the program currently in memory. You can differentiate a changed program from the original version of the program by specifying a new *file-spec*. BASIC saves the program with the new *file-spec*.
2. The program currently in memory does not change.

Examples

```
REPLACE PROGA.NEW
```

25.0 RESEQUENCE (VAX–11 BASIC)

Function

The RESEQUENCE command allows you to resequence the line numbers of the program currently in memory. BASIC also changes all references to the old line numbers so they reference the new line numbers.

Format

```
RESEQUENCE [ lin-num1 [ – lin-num2 ] [ lin-num3 ] ] [ STEP int-const ]
```

Syntax Rules

1. *Lin-num1* is the line number in the program currently in memory where resequencing begins. The default for *lin-num1* is the first line of the program module.
2. *Lin-num2* is the optional end of the range of line numbers to be resequenced. If you specify a range, BASIC begins resequencing with *lin-num1* and resequences through *lin-num2*. If you do not specify *lin-num2*, BASIC resequences the specified line. If you do not specify either *lin-num1* or *lin-num2*, BASIC resequences the entire program.
3. *Lin-num3* specifies the new first line number; the default number for the new first line is 100.
   - If *lin-num3* will cause existing lines to be deleted or surrounded, BASIC signals an error.
   - You can specify *lin-num3*, the new first line number, only when resequencing a range of lines.
4. *Int-const* specifies the numbering increment for the resequencing operation. The default for *int-const* is 10.
5. BASIC signals an error when you try to resequence a program that contains a %IF directive. BASIC also signals an error when you try to resequence a program that has a %INCLUDE directive if the file to be included contains a reference to a line number.

General Rules

1. Before the RESEQUENCE command executes, BASIC verifies the syntax of the program. If the program is not syntactically valid, the RESEQUENCE command does not execute.
2. BASIC sorts the renumbered program in ascending order when the RESEQUENCE command executes.
3. If the renumbering creates a line number greater than the maximum line number of 32767, BASIC signals an error.
4. BASIC signals an error if resequencing causes a change in the order in which program statements are to execute and does not resequence the program.
5. BASIC issues the error "undefined line number" in the case of undefined line numbers and does not resequence the program.

6. BASIC corrects all line numbers for statements that transfer control.
7. BASIC does not modify the program currently in memory when the RESEQUENCE command generates an error.
8. In general, the RESEQUENCE command is not recommended for programs containing error handlers that test the value of ERL. However, the RESEQUENCE command correctly modifies the program if the tests that reference ERL are of this form:

   ```
   ERL relational-operator int-lit
   ```

   The RESEQUENCE command does not correctly renumber programs if the test compares ERL with an expression or a variable, or if ERL follows the relational operator. The following line number references, for example, would not be correctly renumbered:

   ```
   IF ERL = 1000 + A% THEN ...
   IF 1000 > ERL THEN ...
   ```

Examples

```
RESEQUENCE 100-1000 STEP 5
```

26.0 RMSRES (BASIC–PLUS–2)

Function

The RMSRES command allows you to select an RMS memory-resident library to be used when your program is task built. You can also choose to use no RMS memory-resident library. The RMS library supplies RMS code for file and record operations. After you specify a library with the RMSRES command, when you use the BUILD command, *BASIC–PLUS–2* includes the specified library in the Task Builder command file. Every system has an RMS library default set when *BASIC–PLUS–2* is installed.

Format

```
RMSRES lib-param

lib-param:  { file-spec }
            { NONE     }
```

Syntax Rules

1. If you enter the RMSRES command without a *lib-param*, *BASIC–PLUS–2* prompts for one and displays the name of the current default RMS library.

   - *File-spec* can be RMSRES (the RMS memory-resident library) or a user-created resident library. Table 18 lists and describes RMS libraries.
   - NONE tells the Task Builder not to link your task to the RMS default resident library. Therefore, the Task Builder links to the RMS object module library, RMSLIB.OLB.
   - If you type a carriage return in response to the prompt, the current default memory-resident library is used.

General Rules

1. LB: is a RSTS/E logical name for the library account on disk. On *RSX–11M/M–PLUS systems*, you must specify LB:[1,1] before the ODL file name.
2. *BASIC–PLUS–2* links the specified RMS library to your program when you task-build the program. You must use the RMSRES command before you use the BUILD command to include the specified library in the Task Builder command file.
3. If you use an RMS library other than the default, you must specify one of the RMS ODL files listed in Table 18. See the ODLRMS compiler command for more information.

4. The RMSRES library you specify is included in your Task Builder command files until you specify a new library with the RMSRES command or exit from the BASIC environment. When you exit from the environment, the original RMS default library is restored as the default.

5. You can override the RMSRES command with the /RMSRES qualifier added to the BUILD command, but the specified library remains in effect for only one BUILD routine.

6. The Task Builder returns an error message when the requested library is not available.

7. Consult your system manager for information about the RMS libraries available to you. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on using RMS libraries

Table 18: RMS Libraries

Library Name	File Organization			Type of Library	ODL File Required
	Seq	Rel	Ind		
RMSRES	Yes	Yes	Yes	Memory	RMSRLX.ODL
DAPRES	Yes	Yes	Yes	Memory	DAPRLX.ODL
RMSLIB	Yes	Yes	Yes	Disk	RMS11S.ODL RMS12S.ODL RMS11X.ODL RMS12X.ODL DAP11X.ODL

Examples

RSX–11M/M–PLUS Systems

```
RMSRES LB:[1,1]RMSRES
```

RSTS/E Systems

```
RMSRES LB:RMSRES
```

27.0 RUN and RUNNH

Function

The RUN command allows you to execute a program from the BASIC environment without first invoking the *PDP–11* Task Builder or the *VAX–11* Linker to construct an executable image. The program can be:

- A BASIC program brought into memory with the OLD command, created in response to the NEW command, or specified in the RUN command
- An object module or modules placed in memory with the LOAD command
- A combination of the above

RUN executes the program starting at the lowest line number. Program modules previously compiled and placed in memory with the LOAD command are referenced when the RUN command is given. RUNNH executes the program but does not display the program header.

Format

VAX *VAX–11 BASIC*

```
{ RUNNH }
{ RUN   }  [ file-spec ]
```

BP2 *BASIC–PLUS–2*

```
{ RUNNH }
{ RUN   }  [ file-spec ] [ /qualifier ]...
```

Syntax Rules

1. If you specify only the file name, BASIC searches for a file with a BAS (*VAX–11 BASIC*) or B2S (*BASIC–PLUS–2*) file type in the current default directory.
2. If you do not supply a *file-spec*, BASIC executes the program currently in memory.
3. BASIC signals the warning message "No main program" if you do not supply a *file-spec* and do not have a program currently in memory.
4. The RUNNH command is identical to RUN, except that it does not display the program header, current date, and time.

BP2 *BASIC–PLUS–2*

1. */Qualifier* specifies a qualifier keyword that sets a BASIC default. See Section 22.0 for information on BASIC qualifiers. Table 17 lists all *BASIC–PLUS–2* qualifiers and the commands they can be used with, and describes their functions.
2. Support for RUN is an installation option. Use the SHOW command to see whether your system supports the RUN command.

General Rules

1. When you specify a *file-spec* with the RUN command, BASIC brings the program into memory and then executes it. You do not have to bring a program into memory with the OLD command in order to run it. The RUN command executes just as if the program had been brought into memory with the OLD command.
2. If your program calls a subprogram, the subprogram must be compiled and placed in memory with the LOAD command. If your program tries to call a subprogram that has not been compiled and loaded, BASIC signals an error.
3. The RUN command does not create an object module file or a listing file.

VAX–11 BASIC

1. The program stops executing and control passes to the BASIC environment and immediate mode when BASIC encounters a STOP statement in the program.
   - Any BASIC statement that does not require the creation of new storage can be entered in immediate mode to debug the program. You cannot create new variables in immediate mode.
   - Type the CONTINUE command to resume program execution.
2. The RUN command uses whatever qualifiers have been set with the exception of those that have no effect on a program running in the environment. These are:
   - NOCROSS
   - NODEBUG
   - NOLIST
   - NOMACHINE
   - NOOBJECT

   These qualifiers are always in effect when you run a program in the environment.

BASIC–PLUS–2

1. The program stops executing when BASIC encounters a STOP statement:
   - If you used the RUN command to execute the program, BASIC displays a pound sign (#) prompt. In response to the prompt, you can type only CONTINUE to resume program execution, or EXIT to end the program.
   - If you used the RUN/DEBUG command to execute the program, control passes to the *BASIC–PLUS–2* debugger. You can then use *BASIC–PLUS–2* debugger commands to display and change program values and to analyze your program. Use the CONTINUE debugger command to resume program execution. See Part VI in this manual for more information on debugger commands.

Examples

```
RUN PROG1
```

28.0 SAVE

Function

The SAVE command writes the BASIC source program currently in memory to a file on the default or specified device.

Format

```
SAVE [ file-spec ]
```

Syntax Rules

1. If you do not supply a *file-spec*, BASIC saves the file with the name of the program currently in memory and the BAS (*VAX–11 BASIC*) or B2S (*BASIC–PLUS–2*) default file type.
2. If you specify only the file name, BASIC saves the program with the default file type in the current default directory.

General Rules

1. In *BASIC–PLUS–2*, if you type SAVE and the *file-spec* already exists as a disk file, BASIC displays the message "File exists – Rename or Replace".
2. *VAX–11 BASIC* writes a new version of a previously saved program when you type the SAVE command.
3. BASIC stores the sorted program in ascending line number order.
4. You can store the program on a specified device. For example:

   ```
   SAVE DBA1:NEWTST.PRO
   ```

   BASIC saves the file NEWTST.PRO on disk DBA1:.

Examples

```
SAVE JUNK.BAS
```

29.0 SCALE

Function

The SCALE command allows you to control accumulated round-off errors by multiplying numeric values by 10 raised to the scale factor before storing them.

Format

```
SCALE int-const
```

Syntax Rules

1. In *BASIC–PLUS–2*, SCALE with no argument causes BASIC to display the message "Current scale factor is *n*", where *n* is an integer from 0 to 6 inclusive. In *VAX–11 BASIC*, SCALE with no argument causes BASIC to signal the error "illegal argument for command".
2. *Int-const* specifies the power of 10 you want to use as the scaling factor.
   - In *VAX–11 BASIC*, *int-const* must be an integer from 0 to 6, inclusive, or BASIC signals the error "illegal argument for command".
   - In *BASIC–PLUS–2*, *int-const* can be a floating-point or integer number up to 6.999999. BASIC truncates a floating point value and displays the message "%Scale factor has been truncated to *n*", where *n* is the integer portion of the value. If the specified value is greater than 6.999999, BASIC signals the error "Scale factor of *n* is out of range", where *n* is the specified value.

General Rules

1. SCALE affects only values of the data type DOUBLE.
2. BASIC multiplies values using the scale factor you specify. The value 2.488888, for example, is rounded as follows:

Scale:	Produces:
0	2.48889
1	2.4
2	2.48
3	2.488
4	2.4888
5	2.48888
6	2.48889

Examples

```
SCALE 2
```

30.0 SCRATCH

Function

The SCRATCH command clears any program currently in memory, removes any object files loaded with the LOAD command, and resets the program name to NONAME.

Format

```
SCRATCH
```

Syntax Rules

None.

General Rules

None.

Examples

```
SCRATCH
```

31.0 SEQUENCE

Function

The SEQUENCE command causes BASIC to automatically generate line numbers for your program text. BASIC supplies line numbers for your text until you end the procedure or reach the maximum line number of 32767.

Format

```
SEQUENCE [ lin-num ] [ , int-const ]
```

Syntax Rules

1. *Lin-num* specifies the line number where sequencing begins.
   - If you do not specify a *lin-num*, the *VAX–11 BASIC* default is the last line inserted by a SEQUENCE command; if there is no previous SEQUENCE command, the default is line number 100.
   - The *BASIC–PLUS–2* default *lin-num* is always line number 100.
2. *Int-const* specifies the line number increment for your program.
   - If you do not specify an increment, *VAX–11 BASIC* defaults to the *int-const* specified in the last SEQUENCE command; if there is no previous SEQUENCE command, the default is 10.
   - *BASIC–PLUS–2* always defaults to 10.

General Rules

1. If you specify a *lin-num* that already contains a statement, or if the sequencing operation generates a line number that already contains a statement, BASIC signals "Attempt to sequence over existing statement", and returns to normal input mode.
2. Enter your program text in response to the line number prompt; the carriage return ends each line and causes BASIC to generate a new line number.
3. If you enter a CTRL/Z in response to the line number prompt, BASIC terminates the sequencing operation and prompts for another command.
4. You can also terminate the sequence operation in *BASIC–PLUS–2* by typing a carriage return in response to the line number prompt.
5. When the maximum line number of 32767 is reached, BASIC terminates the sequencing process and returns to normal input mode.
6. BASIC does not check syntax during the sequencing process.

Examples

```
SEQUENCE 100,10
```

32.0 SET

Function

The SET command allows you to specify BASIC defaults for all BASIC qualifiers. Qualifiers control the compilation process and the run-time environment. Qualifiers are set or reset as you specify. The defaults you set remain in effect for all subsequent operations until they are reset or until you exit from the compiler.

Format

```
SET  [qualifier,...]
     [/qualifier...]
```

Syntax Rules

1. */Qualifier* specifies a qualifier keyword that sets a BASIC default. See Section 22.0 for information on BASIC qualifiers. Table 16 lists and describes all *VAX–11 BASIC* qualifiers. Table 17 lists and describes all *BASIC–PLUS–2* qualifiers.

VAX

2. If you do not specify any qualifiers, *VAX–11 BASIC* resets all defaults to the defaults specified with the DCL BASIC command.

BP2

3. If you do not specify any qualifiers, *BASIC–PLUS–2* resets all qualifiers except those set with the BRLRES, DSKLIB, LIBRARY, ODLRMS, RMSRES, or EXTEND qualifier to the installation defaults. The SCALE value set with the SCALE command is also not reset to the installation default.

4. *VAX–11 BASIC* signals the error "unknown qualifier" and *BASIC–PLUS–2* signals "Illegal switch" if you do not separate multiple qualifiers with commas or slashes, or if you mix commas and slashes on the same command line. The same error is signaled if you separate qualifiers with a slash but do not prefix the first qualifier with a slash.

General Rules

None.

Examples

```
SET /DOUBLE/BYTE/LIST
```

33.0 SHOW

Function

The SHOW command displays the current defaults for the BASIC compiler on your terminal.

Format

```
SHOW
```

Syntax Rules

None.

General Rules

None.

Examples

VAX–11 BASIC

```
SHOW
VAX-11 BASIC V2.      Current Environment Status    11-DEC-1982 10:05:56.57
DEFAULT DATA TYPE INFORMATION:          LISTING FILE INFORMATION INCLUDES:
    Data type : REAL                        NO Source
    Real size : SINGLE                      NO Cross reference
    Integer size : LONG                        CDD Definitions
    Decimal size : (15,2)                      Environment
    Scale factor : 0                        NO Override of %NOLIST
    NO Round decimal numbers                NO Machine code
                                               Map
COMPILATION QUALIFIERS IN EFFECT:              INCLUDE files
       Object file
       Overflow check integers          FLAGGERS:
       Overflow check decimal numbers          Declining features
       Bounds checking                  NO BASIC PLUS 2 subset
    NO Syntax checking
       Lines
       Variant : 0                      DEBUG INFORMATION:
       Warnings                                Traceback records
       Informational                    NO Debug symbol records
       Setup
       Object Libraries : NONE
Ready
```

BASIC–PLUS–2

```
SHOW
PDP-11 BASIC-PLUS-2 V2.0
ENVIRONMENT INFORMATION:            RMS FILE ORGANIZATION:
    Current edit line : 0              NO Index
    NO Modules loaded                  NO Relative
    NO Main module loaded              NO Sequential
       Run support                     NO Virtual
```

(continued on next page)

```
DEFAULT DATA TYPE INFORMATION:
    Data type : REAL
    Real size : SINGLE
    Integer size : WORD
    Scale factor : 0
COMPILATION QUALIFIERS:
    NO Object
    NO Macro
       Lines
    NO Debug records
    NO Syntax checking
    Flag : Declining
    Variant : 0
LISTING FILE INFORMATION:
   NO Source
   NO Cross Reference
   NO Keywords
      60 lines by 132 columns
BUILD QUALIFIERS:
   NO Dump
   NO Map
   Task extend  : 512
   RMS ODL file : LB:RMSRLX
   BP2 Disk lib : LB:BP2OTS
   BP2 Resident lib : LB:BP2RES
   RMS Resident lib : LB:RMSRES
```

34.0 UNSAVE

Function

The UNSAVE command deletes a specified file from storage.

Format

```
UNSAVE [ file-spec ]
```

Syntax Rules

1. *File-spec* is optional.
   - If you do not supply a *file-spec*, BASIC deletes a file that has the file name of the program currently in memory and a file type of BAS (*VAX–11 BASIC*) or B2S (*BASIC–PLUS–2*).
   - If you do not supply a *file-spec* and do not have a program in memory, BASIC searches for the default file NONAME.BAS.
2. You do not have to supply a full *file-spec*. If you specify only a file name, BASIC deletes the file with the specified name and the BAS (*VAX–11 BASIC*) or B2S (*BASIC–PLUS–2*) file type from the default device and directory. Other file types with the same file name are not deleted.

General Rules

1. The program currently in memory does not change even when it is the deleted file because it is a copy of the deleted file.

Examples

```
UNSAVE DB2:CHECK.DAT
```

PART III
Compiler Directives

1.0 %ABORT

Function

The %ABORT directive terminates program compilation and displays a fatal error message you supply.

Format

```
%ABORT [ str-lit ]
```

Syntax Rules

1. The %ABORT directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %ABORT directive.

General Rules

1. BASIC stops the compilation and terminates the listing file as soon as it encounters a %ABORT directive. *Str-lit* is displayed on the terminal screen and in the compilation listing, if one has been requested.

Examples

```
100     %IF %VARIANT = 2 %THEN
            %ABORT "Cannot compile with variant 2"
        %END %IF
```

2.0 %CROSS

Function

The %CROSS directive causes BASIC to begin or resume accumulating cross-reference information for the listing file.

Format

```
%CROSS
```

Syntax Rules

1. The %CROSS directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %CROSS directive.

General Rules

1. When a cross-reference is requested, the compiler starts or resumes accumulating cross-reference information immediately after encountering the %CROSS directive.

Examples

```
1000     %CROSS
```

3.0 %IDENT

Function

The %IDENT directive lets you identify the version of a program module. The identification text is placed in the object module and printed in the listing header.

Format

```
%IDENT str-lit
```

Syntax Rules

1. *Str-lit* is the identification text. *VAX–11 BASIC* allows *str-lit* to consist of up to 31 ASCII characters. *BASIC–PLUS–2* allows *str-lit* to consist of up to six RAD–50 characters. Both truncate extra characters from *str-lit* and signal a warning message.
2. In *BASIC–PLUS–2*, if *str-lit* contains non–RAD–50 characters, a warning message is issued, and the %IDENT directive is ignored. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on RAD–50 characters. (BP2)
3. The %IDENT directive cannot begin in column one.
4. Only a line number or a comment field can appear on the same physical line as the %IDENT directive.

General Rules

1. The compiler inserts the identification text in the first 6 or 31 character positions of the second line on each listing page. The compiler also includes the identification text in the object module, if the compilation produces one, and in the map file created by the Task Builder (*BASIC–PLUS–2*) or the *VAX–11 Linker*.
2. The %IDENT directive should appear at the beginning of your program if you want the identification text to appear on the first page of your listing. If the %IDENT directive appears after the first program statement, the text will appear on the next page of the listing file.
3. You can use the %IDENT directive only once in a module. If you specify more than one %IDENT directive in a module, BASIC signals a warning and uses the identification text specified in the first %IDENT.

BP2

4. The default *BASIC–PLUS–2* identification text is a 6–digit number. The first two digits represent the compiler base level, while the last four digits represent the month and day. For example, the identification text 100712 represents base level 10, and a date of July 12.

VAX

5. *VAX–11 BASIC* does not provide a default identification text.

Examples

```
100      %IDENT "V3.2"
```

4.0 %IF–%THEN–%ELSE–%END–%IF

Function

The %IF–%THEN–%ELSE–%END–%IF directive lets you conditionally include source code or execute another compiler directive.

Format

```
%IF lex-exp %THEN code [ %ELSE code ] %END %IF
```

Syntax Rules

1. The %IF directive can appear anywhere in a program where a space is allowed, except in column one or within a quoted string. This means that you can use the %IF directive to make a whole statement, part of a statement, or a block of statements conditional.

2. *Lex-exp* is always a LONG integer in *VAX–11 BASIC* and a WORD integer in *BASIC–PLUS–2*. It can be:

   - A lexical constant named in a %LET directive.
   - An integer literal, with or without the percent sign suffix.
   - A lexical built-in function (%VARIANT).
   - Any combination of the above, separated by valid lexical operators. Lexical operators include logical operators, relational operators, and the arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/).

3. *Code* is BASIC program code. It can be any BASIC statement or another compiler directive, including another %IF directive. You can nest %IF directives to eight levels.

4. %THEN, %ELSE, and %END %IF do not have to be on the same physical line as %IF.

General Rules

1. If *lex-exp* is true, BASIC processes the %THEN clause. If *lex-exp* is false, BASIC processes the %ELSE clause. If there is not an %ELSE clause, BASIC processes the %END %IF clause. The compiler includes statements in the %THEN or %ELSE clause in the source program and executes directives in order of occurrence.

2. You must include the %END %IF clause. Otherwise, BASIC assumes the remainder of the program is part of the last %THEN or %ELSE clause and signals the error "missing %END %IF" when compilation ends.

Examples

```
100     %IF (%VARIANT = 2)
        %THEN DECLARE SINGLE HOURLY_PAY(100)
        %ELSE %IF (%VARIANT = 1)
              %THEN DECLARE DOUBLE SALARY_PAY(100)
              %ELSE %ABORT "Can't compile with specified variant"
              %END %IF
        %END %IF
             .
             .
             .
1000    PRINT %IF (%VARIANT = 2)
              %THEN PRINT 'Hourly Wage Chart'
                    GOTO Hourly_routine
              %ELSE PRINT 'Salaried Wage Chart'
                    GOTO Salary_routine
              %END %IF
```

5.0 %INCLUDE

Function

The %INCLUDE directive lets you include BASIC source text from another program file in the current program compilation. *VAX–11 BASIC* also lets you access record definitions in the VAX–11 Common Data Dictionary (CDD).

Format

General

```
%INCLUDE file-spec
```

VAX–11 BASIC

```
%INCLUDE %FROM %CDD str-lit
```

Syntax Rules

1. The %INCLUDE directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %INCLUDE directive.
3. *File-spec* specifies the file to be included. BASIC uses the default device, directory, and file type (BAS in *VAX–11 BASIC* and B2S in *BASIC–PLUS–2*) if you do not specify these parts of the file specification.
4. *File-spec* must be a disk file or BASIC signals an error.
5. *File-spec* must be a string literal enclosed in quotation marks.

VAX–11 BASIC only

1. *Str-lit* specifies a *VAX–11 CDD* path specification. This lets you extract a RECORD definition from the CDD.
2. There are two types of CDD path names: *absolute* and *relative*. An absolute path name begins with CDD$TOP and specifies the complete path to the record definition. A relative path name begins with any string other than CDD$TOP.

General Rules

1. The compiler includes the specified source file in the program compilation at the point of the %INCLUDE directive and prints the included code in the program listing file if the compilation produces one.
2. The included file cannot contain line numbers or BASIC signals the error "Line number may not appear in %INCLUDE file".

3. All statements in the accessed file are associated with the line number of the program line that contains the %INCLUDE directive. This means that a %INCLUDE directive cannot appear before the first line number in a source program.
4. A file accessed by %INCLUDE can itself contain a %INCLUDE directive.
5. All %IF directives in an included file must have a matching %END %IF directive in the file.

VAX–11 BASIC only

1. You can control whether or not included text appears in the compilation listing with the /SHOW:INCLUDE qualifier. When you specify /SHOW:INCLUDE, the compilation listing file identifies any text obtained from an included file by placing a mnemonic in the first character position of the line on which the text appears. The mnemonic is of the form "In" where "I" tells you that the text was accessed with a %INCLUDE directive and "n" is a number that tells you the nesting level of the included text. See the *BASIC User's Guide* for more information on listing mnemonics.
2. When you use the %INCLUDE directive to extract a record definition from the CDD, BASIC translates the CDD definition to the syntax of the BASIC RECORD statement.
3. You can use the /SHOW:CDD_DEFINITIONS to specify that translated CDD definitions (in RECORD statement syntax) are included in the compilation listing file. BASIC places a "C" in column one when the translated RECORD statement appears in the listing file.
4. When you do not specify /SHOW:CDD_DEFINITIONS, BASIC includes the names, data types, and offsets of the CDD record components in the program listing's allocation map.
5. See *BASIC on VAX/VMS Systems* and the *VAX–11 Common Data Dictionary Utilities Reference Manual* for more information on CDD definitions.

Examples

General

```
100      %INCLUDE "YESNO"
```

VAX–11 BASIC only

```
1000      %INCLUDE %FROM %CDD "CDD$TOP.EMPLOYEE"
```

6.0 %LET

Function

The %LET directive declares and provides values for lexical constants. You can use lexical constants only in conditional expressions in the %IF–%THEN–%ELSE directive and in lexical expressions in subsequent %LET directives.

Format

```
%LET %lex-const-nam = lex-exp
```

Syntax Rules

1. *Lex-const-nam* is the name of a lexical constant. Lexical constants are always LONG integers in *VAX–11 BASIC* and WORD integers in *BASIC–PLUS–2*.
2. *Lex-const-nam* must be preceded by a percent sign and cannot end with a dollar sign ($) or percent sign.
3. *Lex-exp* can be:
   - A lexical constant named in a previous %LET directive.
   - An integer literal, with or without the percent sign suffix.
   - A lexical built-in function (%VARIANT)
   - Any combination of the above, separated by valid lexical operators. Lexical operators may be logical operators, relational operators, and the arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/).
4. The %LET directive cannot begin in column one.
5. Only a line number or a comment field can appear on the same physical line as the %LET directive.

General Rules

1. You cannot change the value of *lex-const-nam* within a program unit once it has been named in a %LET directive.

Examples

```
100     %LET %DEBUG_ON = 1%
```

7.0 %LIST

Function

The %LIST directive causes the compiler to start or resume accumulating compilation information for the program listing file.

Format

```
%LIST
```

Syntax Rules

1. The %LIST directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %LIST directive.

General Rules

1. As soon as it encounters the %LIST directive, the compiler starts or resumes accumulating information for the program listing file. Thus, the directive itself appears as the next line in the listing file.
2. The %LIST directive has no effect unless you requested a listing file.

Examples

```
100     %LIST
```

8.0 %NOCROSS

Function

The %NOCROSS directive causes the compiler to stop accumulating cross-reference information for the program listing file.

Format

```
%NOCROSS
```

Syntax Rules

1. The %NOCROSS directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %NOCROSS directive.

General Rules

1. The compiler stops accumulating cross-reference information for the program listing file immediately after encountering the %NOCROSS directive.
2. The %NOCROSS directive has no effect unless you requested cross-reference information.
3. Digital recommends that you not embed a %NOCROSS directive within a statement. Embedding a %NOCROSS directive within a statement makes the accumulation of cross-reference information behave unpredictably.

Examples

```
1000      %NOCROSS
```

9.0 %NOLIST

Function

The %NOLIST directive causes the compiler to stop accumulating compilation information for the program listing file.

Format

```
%NOLIST
```

Syntax Rules

1. The %NOLIST directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %NOLIST directive.

General Rules

1. As soon as it encounters the %NOLIST directive, the compiler stops accumulating information for the program listing file. Thus, the directive itself does not appear in the listing file.
2. The %NOLIST directive has no effect unless you requested a listing file.
3. In *VAX–11 BASIC*, you can override all %NOLIST directives in a program with the /SHOW:OVERRIDE qualifier.

Examples

```
100     %NOLIST
```

10.0 %PAGE

Function

The %PAGE directive causes BASIC to begin a new page in the program listing file immediately after the line that contains the %PAGE directive.

Format

```
%PAGE
```

Syntax Rules

1. The %PAGE directive cannot begin in column one.
2. Only a line number or a comment field can appear on the same physical line as the %PAGE directive.

General Rules

None.

Examples

```
1000      %PAGE
```

11.0 %SBTTL

Function

The %SBTTL directive lets you specify a subtitle for the program listing file.

Format

```
%SBTTL str-lit
```

Syntax Rules

1. *VAX–11 BASIC* allows *str-lit* to contain 45 characters. *BASIC–PLUS–2* allows *str-lit* to contain 48 characters.
2. BASIC truncates extra characters from *str-lit* and does not signal a warning or error.
3. The %SBTTL directive cannot begin in column one.
4. Only a line number or a comment field can appear on the same physical line as the %SBTTL directive.

General Rules

1. The specified subtitle appears underneath the title on the second line of all pages of the listing file until the compiler encounters another %SBTTL or %TITLE directive.
2. Because BASIC assumes that a subtitle is associated with a title, a new %TITLE directive eliminates the current subtitle. In this case, no subtitle appears in the listing until the compiler encounters another %SBTTL directive.
3. If you want a subtitle to appear on the first page of your listing, the %SBTTL directive should appear at the beginning of your program, immediately after the %TITLE directive. Otherwise, the subtitle will appear on the second page of the listing, but not on the first.
4. If you want the subtitle to appear on the page of the listing that contains the %SBTTL directive, the %SBTTL directive should immediately follow a %PAGE directive or a %TITLE directive that follows a %PAGE directive.

Examples

```
100     %SBTTL 'DESMA219 Production Elements'
```

12.0 %TITLE

Function

The %TITLE directive lets you specify a title for the program listing file.

Format

```
%TITLE str-lit
```

Syntax Rules

1. *VAX–11 BASIC* allows *str-lit* to contain 45 characters. *BASIC–PLUS–2* allows *str-lit* to contain 48 characters.
2. BASIC truncates extra characters from *str-lit* and does not signal a warning or error.
3. The %TITLE directive cannot begin in column one.
4. Only a line number or a comment field can appear on the same physical line as the %TITLE directive.

General Rules

1. The specified title appears on the first line of every page of the listing file until BASIC encounters another %TITLE directive in the program.
2. The %TITLE directive should appear on the first line of your program, before the first statement, if you want the specified title to appear on the first page of your listing.
3. If you want the specified title to appear on the page that contains the %TITLE directive, the %TITLE directive should immediately follow a %PAGE directive.
4. Because BASIC assumes that a subtitle is associated with a title, a new %TITLE directive eliminates the current subtitle.

Examples

```
100     %TITLE 'Production Control for DESMA219'
```

13.0 %VARIANT

Function

%VARIANT is a built-in lexical function that allows you to conditionally control program compilation. %VARIANT returns an integer value when you reference it in a lexical expression. You set the variant value with the /VARIANT qualifier when you compile the program or with the SET command.

Format

```
%VARIANT
```

Syntax Rules

1. The %VARIANT function can appear only in a lexical expression.

General Rules

1. The %VARIANT function returns the integer value specified at compile-time with the /VARIANT qualifier to the COMPILE command or with the SET command, or in *VAX–11 BASIC*, set with the DCL BASIC command. The returned integer always has a data type of LONG in *VAX–11 BASIC* and WORD in *BASIC–PLUS–2*.

Examples

```
100     %IF (%LOOP_CONST <= %VARIANT)
        %THEN GOTO Tax_Routine
        %ELSE %ABORT 'Variant too large for program to compile'
        %END %IF
```

PART IV
Statements

1.0 CALL

Function

The CALL statement transfers control to a BASIC subprogram or other callable routine. You can pass optional arguments to the routine and can specify how these arguments are to be passed. When the called routine finishes executing, control returns to the calling program.

Format

```
CALL routine [ pass-mech ] [ ( [ actual-param ],... ) ]

routine:        { sub-nam              }
                { any callable routine }

pass-mech:      { BY REF   }
                { BY VALUE }
                { BY DESC  }

actual-param:   { exp                  }
                { array ( [,]... )     }  [ pass-mech ]
```

Syntax Rules

1. *Routine* is the name of the BASIC SUB subprogram you want to call or the name of any other callable module, such as a system service or an RTL routine on *VAX/VMS systems*. It cannot be a variable name.

2. *Pass-mech* specifies how arguments are passed to the called routine. If you do not specify a *pass-mech*, BASIC passes arguments as indicated in Tables 19 and 20.
3. You can use passing mechanisms only when calling non–BASIC routines.
4. When *pass-mech* appears before the parameter list, it applies to all arguments passed to the called *routine*. You can override this passing mechanism by specifying a *pass-mech* for individual arguments in the *actual-param* list.
5. *Actual-param* lists the arguments to be passed to the called *routine*.
6. You can pass expressions or entire arrays. Optional commas in parentheses after the array name specify the dimensions of the array. The number of commas is equal to the number of dimensions minus one. Thus, no comma specifies a one-dimensional array, one comma specifies a two-dimensional array, two commas specify a three-dimensional array, and so on.

VAX–11 BASIC

1. The name of the *routine* can consist of from 1 to 31 characters and must conform to the following rules:
   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), periods (.), or underscores (_).
   - A quoted name can consist of any combination of printable ASCII characters.
2. *Routine* can be a system service, an RTL routine, or any procedure written in a language that supports the *VAX–11 Procedure Calling Standard*. See *BASIC on VAX/VMS Systems* for more information on using system services, RTL routines, and other procedures.
3. *VAX–11 BASIC* allows you to pass up to 255 parameters.
4. You cannot pass virtual arrays.

BASIC–PLUS–2

1. The name of the *routine* can consist of from one to six characters and must conform to the following rules:
   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).
   - A quoted name can consist of any combination of alphabetic characters, digits, dollar signs ($), periods (.), or spaces.
2. *Routine* can be a *BASIC–PLUS–2* subprogram or a subprogram written in another language.

Note

Although you can call routines written in other languages, *BASIC–PLUS–2* does not support calling anything but *BASIC–PLUS–2* routines.

3. You can pass all arguments BY REF, but you can pass only string values and entire arrays BY DESC.

4. *BASIC–PLUS–2* lets you pass up to eight parameters to a *BASIC–PLUS–2* subprogram and up to 255 parameters to a MACRO–11 subprogram.

General Rules

1. The optional *pass-mech* clauses tell BASIC how to pass arguments to the called subprogram. Table 19 describes *VAX–11 BASIC* parameter passing mechanisms. Table 20 describes *BASIC–PLUS–2* parameter passing mechanisms.

   - BY REF specifies that BASIC passes the argument's address. This is the default for all arguments except strings and entire arrays.
   - BY VALUE specifies that *VAX–11 BASIC* passes the argument's 32–bit value and that *BASIC–PLUS–2* passes the argument's 16–bit value.
   - BY DESC specifies that BASIC passes the address of a *VAX–11 BASIC* descriptor or a *BASIC–PLUS–2* descriptor. For information about the format of a *VAX–11 BASIC* descriptor for strings and arrays, see Appendix C in *BASIC on VAX/VMS Systems*; for information on other types of descriptors, see the *VAX Architecture Handbook*. *BASIC–PLUS–2* creates descriptors only for strings and arrays; these descriptors are described in Appendix C in *BASIC on RSX–11M/M–PLUS Systems* and *BASIC on RSTS/E Systems*.

2. You can specify a null argument as an *actual-param* for non–BASIC routines by omitting the argument and the *pass-mech*, but not the commas or parentheses. This forces BASIC to pass a null argument as defined by your operating system and allows you to access system routines from BASIC.

3. Arguments in the *actual-param* list must agree in data type and number with the formal parameters specified in the subprogram.

4. An argument is modifiable when changes to it are evident in the calling program. Changing a modifiable parameter in a subprogram means the parameter is changed for the calling program as well. Variables and entire arrays passed BY DESC or BY REF are modifiable.

5. An argument is nonmodifiable when changes to it are not evident in the calling program. Changing a nonmodifiable argument in a subprogram does not affect the value of that argument in the calling program. Arguments passed BY VALUE, constants, and expressions are nonmodifiable. Passing an argument as an expression (by placing it in parentheses) changes it from a modifiable to a nonmodifiable argument.

6. For expressions and virtual array elements passed BY REF, BASIC makes a local copy of the value, and passes the address of this local copy. For dynamic string arrays, BASIC passes a descriptor of the array of string descriptors. BASIC passes the address of the argument's actual value for all other arguments passed BY REF.

7. No files are closed when the CALL statement executes.

VAX–11 BASIC

1. Only BYTE, WORD, LONG, and SINGLE values can be passed by BY VALUE. BYTE and WORD values passed by VALUE are converted to LONG values.

(BP2) *BASIC–PLUS–2*

1. Only BYTE and WORD values can be passed BY VALUE. BYTE values passed BY VALUE are converted to WORD values.
2. *BASIC–PLUS–2* does not allow recursion. That is, once a subprogram is called, it cannot be called again until the SUBEND or SUBEXIT statement has executed or until an error has been trapped with ON ERROR GO BACK.

Examples

```
200     CALL SUB1 BY REF (EMPNAME$, (Z%) BY VALUE, D$() BY DESC)
```

Table 19: VAX–11 BASIC Parameter Passing Mechanisms

Argument Type	BY VALUE	BY REF	BY DESC
Numeric Arguments			
Variables	**YES	*YES	YES
Constants	**YES	*Local copy	Local copy
Expressions	**YES	*Local copy	Local copy
Array elements	**YES	*YES	YES
Virtual array elements	**YES	*Local copy	Local copy
Entire arrays	NO	YES	*YES
Entire virtual arrays	NO	NO	NO
String Arguments			
Variables	NO	YES	*YES
Constants	NO	Local copy	*Local copy
Expressions	NO	Local copy	*Local copy
Array elements	NO	YES	*YES
Virtual array elements	NO	Local copy	*Local copy
Entire arrays	NO	YES	*YES
Entire virtual arrays	NO	NO	NO

* One asterisk indicates the default parameter passing mechanisms for BASIC programs.

** Two asterisks indicate that the value can have 32 bits, at most.

Table 20: BASIC–PLUS–2 Parameter Passing Mechanisms

Argument Type	BY VALUE	BY REF	BY DESC
Numeric Arguments			
Variables	**YES	*YES	NO
Constants	**YES	*Local copy	NO
Expressions	**YES	*Local copy	NO
Array elements	**YES	*Local copy	NO
Virtual array elements	**YES	*Local copy	NO
Entire arrays	NO	YES	*YES
Entire virtual arrays	NO	NO	*YES
String Arguments			
Variables	NO	YES	*YES
Constants	NO	Local copy	*Local copy
Expressions	NO	Local copy	*Local copy
Array elements	NO	Local copy	*Local copy
Virtual array elements	NO	Local copy	*Local copy
Entire arrays	NO	YES	*YES
Entire virtual arrays	NO	NO	*YES

* One asterisk indicates the default parameter passing mechanisms for BASIC programs. You should never use a BY clause when calling a BASIC subprogram from a BASIC main program.

** Two asterisks indicate that the value can be only WORD or BYTE. Other data types require more than the 16 bits of storage allowed.

Note

DIGITAL recommends that you not pass entire virtual arrays as parameters in the CALL statement. Instead, you can share the data in a virtual array between a calling program and a subprogram by opening a virtual file in either program and dimensioning the array (using the same channel number) in both programs.

2.0 CHAIN

Note

The CHAIN statement is not recommended for new program development. DIGITAL recommends that you use the CALL statement for program segmentation.

Function

The CHAIN statement transfers control from the current program to an executable BASIC program. CHAIN closes all files, then requests that the new program begin execution. Control does not return to the original program when the new program finishes executing.

Format

General

```
CHAIN str-exp
```

BASIC–PLUS–2 on RSTS/E only

```
CHAIN str-exp [ LINE lin-num ]
```

Syntax Rules

1. *Str-exp* represents the file specification of the program to which control is passed. It can be a quoted or unquoted string.
   - *Str-exp* must refer to an executable image or BASIC signals an error.
   - If you do not specify a file type, *VAX–11 BASIC* searches for an EXE file type and *BASIC–PLUS–2* searches for a TSK file type.
   - You cannot chain to a program on another node.

BASIC–PLUS–2

RSTS

1. On *RSTS/E systems* you can specify that control pass to a specified line number in another *BASIC–PLUS–2* program.
   - *Lin-num* specifies a line in another BASIC program. It must be in the range 1 to 32767, inclusive.
   - If you specify a *lin-num*, the program to which control passes must have been compiled with the /CHAIN qualifier. The /CHAIN qualifier overrides the /NOLINE qualifier.

General Rules

1. Execution starts at the first line number of the specified program unless your system is *RSTS/E* and you have specified a *lin-num* at which execution is to start.

RSTS

2. On *RSTS/E systems, BASIC–PLUS–2* signals an error when the specified line number does not exist.

3. Before chaining takes place, all active output buffers (except terminal-format files) are written, all open files are closed, and all storage is released. On *RSTS/E* systems, the last buffer (512 bytes) of a terminal-format file does not get written unless the file is closed before the CHAIN statement executes.
4. Because a CHAIN statement passes control from the executing image, the values of any program variables are lost. This means that you can pass parameters to a chained program only by using files or a system-specific feature such as the GET/PUT Core Common on *RSTS/E systems*, or LIB$GET and LIB$PUT on *VMS systems*.
5. See *BASIC on RSTS/E Systems* or *BASIC on RSX–11M/M–PLUS Systems* for information about how the CHAIN statement is implemented on your system.

Examples

General

```
100     CHAIN "PROG2"

900     CHAIN PROG5.EXE
```

BASIC–PLUS–2 on RSTS/E only

```
200     CHAIN PROGA.TSK LINE 300
```

3.0 CHANGE

Function

The CHANGE statement: 1) converts a string of characters to their ASCII integer values or 2) converts a list of numbers to a string of ASCII characters.

Format

String Variable to Array

```
CHANGE str-exp TO num-array
```

Array to String Variable

```
CHANGE num-array TO str-vbl
```

Syntax Rules

1. *Num-array* should be a one-dimensional array (or list). If you specify a two-dimensional array, BASIC converts only the zero row of that array. BASIC does not support CHANGE to or from arrays of more than two dimensions.
2. *Str-exp* is a string expression.
3. VAX *VAX–11 BASIC* does not support RECORD elements as a destination string or as a source or destination array for the CHANGE statement.

General Rules

String Variable to Array

1. This format converts each character in *str-exp* to its ASCII value.
2. BASIC assigns the value of *str-exp*'s length to the zero element (0) or (0,0) of the *num-array*.
3. BASIC assigns the ASCII value of the first character in *str-exp* to the first element, (1) or (0,1), of *num-array*, the ASCII value of the second character to the second element, (2) or (0,2), and so on.
4. If the string is longer than the bounds of *num-array*, BASIC does not translate the excess characters, and signals the error "subscript out of range" (ERR=55). Element zero, (0) or (0,0), of *num-array* still contains the length of *str-exp*.

Array to String Variable

1. This format converts the elements of *num-arr* to a string of characters.
2. The length of *str-vbl* is determined by the value in the zero element, (0) or (0,0), of *num-array*. If the value of element zero is greater than the array bounds, BASIC signals the error "subscript out of range" (ERR=55).

3. BASIC changes the first element, (1) or (0,1), of *num-array* to its ASCII character equivalent, the second element, (2) or (0,2), to its ASCII equivalent, and so on. The length of the returned string is determined by the value in the zero element of *num-array*. For example, if *num-arr* is dimensioned as (10), but the zero element (0) contains the value 5, BASIC changes only elements (1), (2), (3), (4), and (5) to string characters.
4. BASIC truncates floating-point values to integers before converting them to characters.
5. Values in array elements are treated modulo 256.

Examples

String Variable to Array

```
50      DIM ARRAY_CHANGES%(6)
60      CHANGE "ABCDE" TO ARRAY_CHANGES%
```

Array to String Variable

```
200     CHANGE ARRAY_CHANGES% TO A$
```

4.0 CLOSE

Function

The CLOSE statement ends I/O processing to a device or file on the specified channel.

Format

```
CLOSE chnl-exp,...
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It can be preceded by an optional pound sign (#).

General Rules

1. BASIC writes the contents of any active buffers to the file or device before it closes that file or device.
2. Channel 0 (the controlling terminal) cannot be closed. An attempt to do so has no effect.
3. If you close a magnetic tape file that is open FOR OUTPUT, BASIC writes an end-of-file on the magnetic tape.
4. If you try to close a channel that is not currently open, BASIC does not signal an error and the CLOSE statement has no effect.

Examples

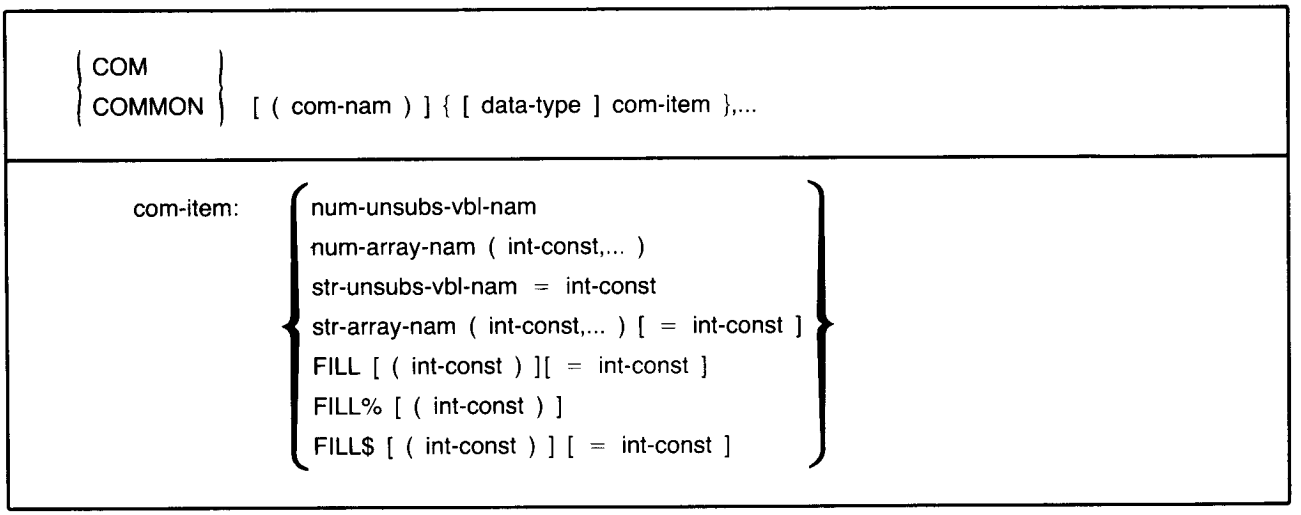
```
1000      CLOSE #1, 3
```

5.0 COMMON

Function

The COMMON statement defines a named, shared storage area called a COMMON block or program section (PSECT). BASIC program modules can access the values stored in the COMMON by specifying a COMMON with the same name.

Format



Syntax Rules

1. *Com-nam* is optional. If present, it must be in parentheses.
2. A COMMON can have the same name as a program variable. However, in *BASIC–PLUS–2*, a COMMON cannot have the same name as a subprogram within the same task image.
3. A COMMON and a MAP in the same program module cannot have the same name.
4. *Com-item* declares the name and format of the data to be stored.

   - *Num-unsubs-vbl-nam* and *num-arr-nam* specify a numeric variable or a numeric array.
   - *Str-unsubs-vbl-nam* and *str-arr-nam* specify a fixed-length string variable or array. You can specify the number of bytes to be reserved for the variable with the *=int-const* clause. The default string length is 16.
   - The FILL, FILL%, and FILL$ keywords allow you to reserve parts of the record buffer within or between data elements and to define the format of the storage. *Int-const* specifies the number of FILL items to be reserved. The *=int-const* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 21 describes FILL item format and storage allocation.

Note

In the applicable formats of FILL, *(int-const)* represents a repeat count, not an array subscript. FILL (n) represents n elements, not n + 1.

5. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.

6. When you specify a *data-type*, all following *com-items*, including FILL items, are of that data type until you specify a new data type.

7. If you do not specify any *data-type*, *com-items* take the current default data type and size.

8. Variable names, array names, and FILL items following a *data-type* cannot end in a dollar sign or percent sign character.

9. Variables and arrays declared in a COMMON statement cannot be declared elsewhere in the program by any other declarative statements.

10. COMMON elements must be separated with commas.

VAX *VAX–11 BASIC*

1. The default *com-nam* is "$BLANK".

2. *Com-nam* can consist of from 1 to 31 characters. The first character of the name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), periods (.), or underscores (_).

BP2 *BASIC–PLUS–2*

1. The default *com-nam* is ".$$$$.".

2. *Com-nam* can consist of from one to six characters. The first character must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).

Table 21: FILL Item Formats and Storage Allocations

FILL Format	Storage Allocation
FILL	Allocates storage for one floating-point element unless preceded by a *data-type*; the number of bytes allocated depends on the default floating-point data size or the specified *data-type*.
FILL(int-const)	Allocates storage for the number of floating-point elements specified by *int-const* unless preceded by a *data-type*; the number of bytes allocated for each element depends on the default floating-point data size or the specified *data-type*.
FILL%	Allocates storage for one integer element; the number of bytes allocated depends on the default integer size.
FILL%(int-const)	Allocates storage for the number of integer elements specified by *int-const*; the number of bytes allocated for each element depends on the default integer size.
FILL$	Allocates 16 bytes of storage for a string element. The dollar sign can be omitted if the FILL keyword is preceded by the STRING *data-type*.
FILL$(int-const)	Allocates 16 bytes of storage for the number of string elements specified by *int-const*. The dollar sign can be omitted if the FILL keyword is preceded by the STRING *data-type*.
FILL$ = int-const	Allocates the number of bytes of storage specified by = *int-const* for a string element. The dollar sign can be omitted if the FILL keyword is preceded by the STRING *data-type*.
FILL$(int-const) = int-const	Allocates the number of bytes of storage specified by = *int-const* for the number of string elements specified by *int-const*. The dollar sign can be omitted if the FILL keyword is preceded by the STRING *data-type*.

Note

In the applicable formats of FILL, *(int-const)* represents a repeat count, not an array subscript. FILL (n) represents n elements, not n + 1.

General Rules

1. A COMMON area and a MAP area with the same name, in different program modules, specify the same storage area.
2. BASIC does not execute COMMON statements. The COMMON statement allocates and defines the data storage area at compile time.
3. When you link or task-build your program, the size of the COMMON area is the size of the largest COMMON area with that name. That is, BASIC concatenates COMMON statements with the same *com-nam* within a single program module into a single PSECT. The total space allocated is the sum of the space allocated in the concatenated COMMON statements.
4. The COMMON statement must lexically precede any reference to variables declared in it.
5. A COMMON area can be accessed by more than one program module, as long as you define the *com-nam* in each module that references the COMMON.

6. Variable names in a COMMON statement in one program module need not match those in another program module.
7. *VAX–11 BASIC* does not initialize variables in COMMON blocks.
8. Since *BASIC–PLUS–2* initializes variables in COMMON blocks, you must use unique names for each variable in each COMMON block.
9. (BP2) In *BASIC–PLUS–2*, you should know how your program overlays if data stored in a COMMON area is to be shared by several program modules. The COMMON should be named in an overlay unit that will remain in memory as long as program units need to reference the COMMON data. If the overlay that names the COMMON is forced out of memory, BASIC reinitializes the COMMON area to zero when the overlay is brought back into memory. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on overlay structures.
10. The *data-type* specified for *com-items* or the default data type and size determines the amount of storage reserved in a COMMON block:
    - BYTE integers reserve 1 byte.
    - WORD integers reserve 2 bytes.
    - LONG integers reserve 4 bytes.
    - SINGLE floating-point numbers reserve 4 bytes.
    - DOUBLE floating-point numbers reserve 8 bytes.
    - GFLOAT floating-point numbers reserve 8 bytes (*VAX–11 BASIC* only).
    - HFLOAT floating-point numbers reserve 16 bytes (*VAX–11 BASIC* only).
    - DECIMAL(d,s) packed decimal numbers reserve (d+1)/2 bytes (*VAX–11 BASIC* only).
    - STRING reserves 16 bytes (the default) or the number of bytes you specify with *=int-const*.

Examples

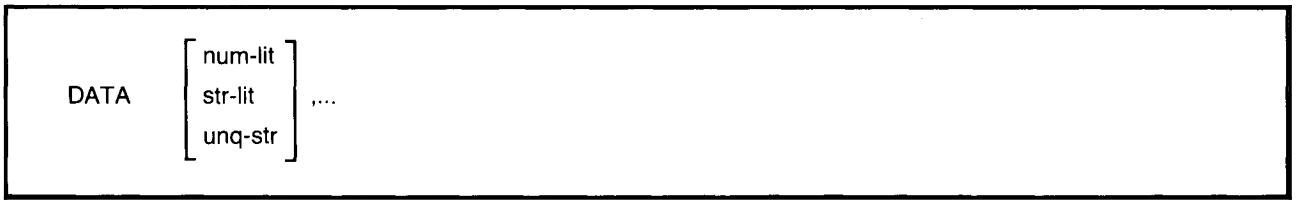
```
500     COMMON (INVEN) INTEGER SHELF_NUMBER, STRING ROW = 2, &
                DOUBLE FILL, PART_BIN, LIST_PRICE
```

6.0 DATA

Function

The DATA statement creates a data block for the READ statement.

Format



Syntax Rules

1. *Num-lit* specifies a numeric literal.
2. *Str-lit* is a character string that starts and ends with double or single quotation marks. The quotation marks must match.
3. *Unq-str* is a character sequence that does not start and end with double or single quotation marks and does not contain a comma.
4. Commas separate data elements. If a comma is part of a data item, the entire item must be enclosed in quotation marks.
5. Because BASIC treats comment fields in DATA statements as part of the DATA sequence, do not include comments.
6. A DATA statement must be the last or the only statement on a line.
7. DATA statements must end with a line terminator. BASIC interprets all characters except the ampersand (&) between the keyword DATA and the final line terminator as part of the data. You can continue DATA statements by placing an ampersand (&) immediately before the line terminator.
8. You cannot use the percent sign suffix for integer constants that appear in DATA statements. An attempt to do so causes BASIC to signal "Data format error" (ERR=50) when you try to run the program.

General Rules

1. DATA statements are local to a program module.
2. BASIC does not execute DATA statements. Instead, BASIC passes control to the next executable statement.
3. A program can have more than one DATA statement. BASIC assembles data from all DATA statements in a single program unit into a lexically ordered single data block.
4. BASIC ignores leading and trailing blanks and tabs unless they are in a string literal.

5. Commas are the only valid data delimiters. You must use a quoted string literal if the comma is to be part of a string.
6. BASIC ignores DATA statements without an accompanying READ statement.
7. BASIC signals the error "Data format error" if the DATA item does not match the data type of the variable specified in the READ statement or if a data element that is to be read into an integer variable ends with a percent sign (%). If a string data element ends with a dollar sign ($), BASIC treats the dollar sign as part of the string.

Examples

```
300     DATA 35, 32.3, PRODUCTION SEQUENCE, 'SYSTEM', '1,2'
```

7.0 DECLARE

Function

The DECLARE statement explicitly assigns a data type to and names a variable, an entire array, a function, or a constant.

Format

Variables

```
DECLARE data-type decl-item [, [ data-type ] decl-item ]...
```

DEF Functions

```
DECLARE data-type FUNCTION { def-nam [ ( [ def-param ],... ) ] },...
```

Named Constants

```
DECLARE data-type CONSTANT { const-nam = const },...
```

```
decl-item:    { unsubs-vbl-nam
              { array-nam ( int-const,... )

def-param:    [ data-type ]
```

Syntax Rules

1. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.

Variables

1. *Decl-item* names a variable or an array.
2. A *decl-item* cannot end in a percent sign (%) or dollar sign ($).
3. A *decl-item* named in a DECLARE statement cannot also be named in another DECLARE statement, or a DEF, EXTERNAL, FUNCTION, SUB, COMMON, MAP, or DIM statement.
4. *Int-const* specifies the upper bounds of the *array-nam*.
5. Each *decl-item* is associated with the preceding *data-type*. A *data-type* is required for the first *decl-item*.
6. *Decl-items* of data-type STRING are dynamic strings.

DEF Functions

1. *Def-nam* names the DEF function. It cannot end with a percent sign (%) or dollar sign ($).
2. *Data-type* specifies the data type of the value the function returns.
3. *Def-params* specify the number and, optionally, the *data-type* of the DEF parameters. Parameters define the arguments the DEF expects to receive when invoked.

   - When you specify a *data-type*, all following parameters are of that data type until you specify a new data type.
   - If you do not specify any *data-type*, parameters take the current default data type and size.
   - The number of parameters equals the number of commas plus one. For example, empty parentheses specify one parameter of the default type and size; one comma inside the parentheses specifies two parameters of the default type and size, and so on. One *data-type* inside the parentheses specifies one parameter of the specified data type; two *data-types* separated by one comma specifies two parameters of the specified type, and so on.

Named Constants

1. *Const-nam* is the name you assign to the *const*.
2. *Data-type* specifies the data type of the *const-nam*. The value of the *const* must be numeric if the data type is numeric and string if the data type is STRING. If the *data-type* is STRING, *const* must be a quoted string or another string constant.
3. *Const* cannot end with a percent sign (%) or a dollar sign ($).
4. *Const* cannot be of the RFA data type.
5. For *VAX–11 BASIC*, string constants cannot exceed 498 characters.
6. For *BASIC–PLUS–2*, string constants cannot exceed 128 characters.
7. *VAX–11 BASIC* allows *const* to be an expression for all data types except DECIMAL. Expressions are not allowed as values when you name DECIMAL constants.
8. *BASIC–PLUS–2* allows *const* to be an expression for STRING and INTEGER data types. Expressions are not allowed as values when you name floating-point constants.
9. Allowable operators in DECLARE CONSTANT expressions include all valid arithmetic, relational, and logical operators except exponentiation. Built-in functions cannot be used in DECLARE CONSTANT expressions. The following examples use valid expressions as values:

```
100     DECLARE DOUBLE CONSTANT MAX_VALUE = (PI / 2)(VAX-11 BASIC only)
100     DECLARE STRING CONSTANT LEFT_ARROW = ('<-----' + LF + CR)
```

General Rules

1. The DECLARE statement is not executable.
2. The DECLARE statement must lexically precede any reference to the variables, functions, or constants named in it.
3. You cannot declare virtual arrays.
4. To avoid confusion and to retain BASIC's implicit data typing feature, variable names ending with a dollar sign or percent sign are invalid in a DECLARE statement.

Variables

1. Variables named in a DECLARE statement are initialized to zero if numeric or to the null string if string.
2. Subsequent *decl-items* are associated with the specified data type until you specify another *data-type*.

DEF Functions

1. The DECLARE FUNCTION statement allows you to name a function defined in a DEF statement, specify the data type of the value the function returns, and declare the number and data type of the DEF parameters.
2. *Data-type* keywords must be separated by commas. For example:

```
100     DECLARE DOUBLE FUNCTION INTEREST(,,DOUBLE,SINGLE)
```

   This example decares two parameters of the default type and size, one DOUBLE parameter, and one SINGLE parameter for the function named INTEREST.

3. The first specification of a *def-param* is the default for subsequent arguments until you specify another *def-param*.

Named Constants

1. The DECLARE CONSTANT statement allows you to name a constant value and assign a data type to that value. Note that you can specify only one data type in a DECLARE CONSTANT statement. To declare another constant, you must use a second DECLARE CONSTANT statement.
2. You cannot change the value assigned to *const-nam*.
3. You cannot use a *const-nam* where a variable is required.

4. In *VAX–11 BASIC*, the specified *data-type* determines the data type of *const*. For example: 


```
100     DECLARE WORD CONSTANT MMM = 1.5
200     DECLARE REAL CONSTANT ZZZ = 123%
300     DECLARE BYTE CONSTANT YYY = '123'L
400     PRINT MMM,ZZZ,YYY

RUNNH

 1              123             123
```

In this example, BASIC truncates the value 1.5 to a WORD integer, and ignores the percent suffix and the L (LONG) data type.

5. *BASIC–PLUS–2* signals the error "Constant is inconsistent with the type of <name> " if the data type of *const* does not match the specified *data-type*. 

Note

Data types specified in a DECLARE statement override any defaults specified in COMPILE command qualifiers or OPTION statements.

Examples

Variables

```
100     DECLARE INTEGER CATALOG_NUM, DOUBLE PRICE, STRING ITEM_NAME
```

DEF Functions

```
100     DECLARE INTEGER FUNCTION AMOUNT(,,DOUBLE,BYTE,,)
```

Named Constants

```
100     DECLARE DOUBLE CONSTANT INTEREST_RATE = 15.22
```

8.0 DEF

Function

The DEF statement lets you define a single- or multi-line function.

Format

Single-Line DEF

```
DEF [ data-type ] def-nam [ ( [ [ data-type ] unsubs-vbl-nam ],... ) ] = exp
```

Multi-Line DEF

```
DEF [ data-type ] def-nam [ ( [ [ data-type ] unsubs-vbl-nam ],... ) ]
        [ statement ]...
{ END DEF }
{ FNEND   }
```

Syntax Rules

1. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.
2. The *data-type* that precedes the *def-nam* specifies the data type of the value returned by the DEF function.
3. *Def-nam* is the name of the DEF function. The *def-nam* may contain from 1 to 31 characters.
4. If the *def-nam* also appears in a DECLARE FUNCTION statement, the following rules apply:
   - A function *data-type* is required.
   - The first character of the *def-nam* must be an alphabetic character (A through Z). The remaining characters may be any combination of letters, digits (0 through 9), dollar signs ($), underscores (_), or periods (.), with one restriction: the last character cannot be a dollar sign.
5. If the *def-nam* does not appear in a DECLARE FUNCTION statement, but the DEF statement appears before the first reference to the *def-nam*, the following rules apply:
   - The function *data-type* is optional.
   - The first character of the *def-nam* must be an alphabetic letter (A through Z). The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods.

(continued on next page)

- If a function *data-type* is specified, the last character in the *def-nam* cannot be a dollar sign or percent sign.

- If a function *data-type* is not specified, the last character in the *def-nam* must be a percent sign (%) for an INTEGER function, a dollar sign ($) for a STRING function, or a letter, digit, period, or underscore for a function of the default type and size.

6. If the *def-nam* does not appear in a DECLARE FUNCTION statement, and the DEF statement appears after the first reference to the *def-nam*, the following rules apply:

   - The function *data-type* cannot be present.

   - The first two characters of the *def-nam* must be FN. The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods, with one restriction: the last character must be a percent sign (%) for an INTEGER function, a dollar sign ($) for a STRING function, or a letter, digit, period, or underscore for a function of the default type and size.

   - There must be at least one character between the FN characters and the ending dollar sign or percent character. FN$ and FN% are not valid function names.

7. *Unsubs-vbl-nam* specifies optional formal DEF parameters. Because the parameters are local to the DEF function, any reference to these variables outside the DEF body creates a different variable.

8. You can specify the *data-type* of DEF parameters with a data-type keyword or, in *VAX–11 BASIC*, with a data type defined in a RECORD statement. If you do not include a data type, the parameters are of the default type and size. Parameters that follow a data-type keyword are of the specified type and size until you specify another data type.

9. *BASIC–PLUS–2* allows you to specify up to eight parameters in a DEF statement.

10. *VAX–11 BASIC* allows you to specify up to 255 parameters in a DEF statement.

Single-Line DEF

1. *Exp* specifies the operations the function performs.

Multi-Line DEF

1. *Statements* specify the operations the function performs.

2. The END DEF or FNEND statement is required to end a multi-line DEF.

3. You can use any BASIC statement except END FUNCTION, END SUB, FUNCTION, FUNCTIONEND, FUNCTIONEXIT, DEF, or DEF* in a function definition.

General Rules

1. When BASIC encounters a DEF statement, control of the program passes to the next executable statement after the DEF.

2. Functions are invoked when you use the function name in an expression.

3. You cannot specify how parameters are passed. When you invoke a function, BASIC evaluates parameters from left to right and passes parameters to the function so that they cannot be modified. Numeric parameters are passed BY VALUE and string parameters are passed BY DESC, where the descriptor points to a local copy. DEF functions may reference variables in the main program, but they cannot reference variables in other DEF or DEF* functions. A DEF function may, therefore, modify other variables in the program, but not variables within another DEF function.

4. A DEF is local to the program or subprogram that defines it.

5. The DEF statement, or the first invocation of a function, whichever occurs first, constitutes the declaration of the function. The DECLARE FUNCTION statement defines the name of the function, but does not invoke it.

6. If your program invokes a function with a name that does not start with FN before the DEF statement defines the function, or if the number of parameters, types of parameters, or type of result declared in the invocation disagree with the number or types of parameters defined in the DEF statement, BASIC signals an error.

7. DATA statements in a multi-line DEF are not local to the function; they are local to the program module containing the function definition.

8. The function value (that is, the location storing the value the function returns) is initialized to zero or the null string each time you invoke the function.

9. ON ERROR GO BACK is the default error handler in a DEF function definition.

10. ON ERROR statements within a DEF are local to the function.

11. A GOTO, GOSUB, ON ERROR GOTO, or RESUME statement in a multi-line function definition must refer to a line number or label in the same function definition.

12. You cannot transfer control into a multi-line DEF except by invoking the function.

13. DEF functions can be recursive.

Examples

Single-Line DEF

```
1000     DEF DOUBLE ADD (DOUBLE A, B, SINGLE C, D, E) = A + B + C + D + E
2000     INPUT 'Enter five numbers to be added';V,W,X,Y,Z
2010     PRINT 'The sum is';ADD(V,W,X,Y,Z)
```

Multi-Line DEF

```
1000     DEF DOUBLE PAYROLL(INTEGER HOURS, REAL RATE)
             EXIT DEF IF HOURS = 0
             DECLARE INTEGER OVERTIME
             OVERTIME = HOURS - 40
             IF OVERTIME <= 0
                THEN HOURS = HOURS
                ELSE HOURS = 40
             END IF
             DECLARE REAL CONSTANT OVER_RATE = 1.5
             PAYROLL = (HOURS * RATE) + (OVERTIME * (OVER_RATE * RATE))
1030     END DEF
1040     INPUT "Your hours this week";MY_HOURS
1045     INPUT "Your pay rate";MY_PAY_RATE
1050     PRINT 'Your pay for the week is';PAYROLL(MY_HOURS,MY_PAY_RATE)
```

9.0 DEF*

Function

The DEF* statement lets you define a single- or multi-line function.

Note

The DEF* statement is not recommended for new program development. DIGITAL recommends that you use the DEF statement for defining single- and multi-line functions.

Format

*Single-Line DEF**

```
DEF* [ data-type ] def-nam [ ( [ [ data-type ] unsubs-vbl-nam ],... ) ] = exp
```

*Multi-Line DEF**

```
DEF* [ data-type ] def-nam [ ( [ [ data-type ] unsubs-vbl-nam ],... ) ]
    [ statement ]...
{ END DEF }
{ FNEND   }
```

Syntax Rules

1. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.
2. The *data-type* that precedes the *def-nam* specifies the data type of the value returned by the DEF* function.
3. *Def-nam* is the name of the DEF* function. The *def-nam* may contain from 1 to 31 characters.
4. If the *def-nam* also appears in a DECLARE FUNCTION statement, the following rules apply:
   - A function *data-type* is required.
   - The first character of the *def-nam* must be an alphabetic character (A through Z). The remaining characters may be any combination of letters, digits (0 through 9), dollar signs ($), underscores (_), or periods (.), with one restriction: the last character cannot be a dollar sign.

5. If the *def-nam* does not appear in a DECLARE FUNCTION statement, but the DEF* statement appears before the first reference to the *def-nam*, the following rules apply:

   - The function *data-type* is optional.
   - The first character of the *def-nam* must be an alphabetic letter (A through Z). The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods.
   - If a function *data-type* is specified, the last character in the *def-nam* cannot be a dollar sign or a percent sign.
   - If a function *data-type* is not specified, the last character in the *def-nam* must be a percent sign (%) for an INTEGER function, a dollar sign ($) for a STRING function, or a letter, digit, period, or underscore for a function of the default type and size.

6. If the *def-nam* does not appear in a DECLARE FUNCTION statement, and the DEF* statement appears after the first reference to the *def-nam*, the following rules apply:

   - The function *data-type* cannot be present.
   - The first two characters of the *def-nam* must be FN. The remaining characters can be any combination of letters, digits, dollar signs, underscores, or periods, with one restriction: the last character must be a percent sign (%) for an INTEGER function, a dollar sign ($) for a STRING function, or a letter, digit, period, or underscore for a function of the default type and size.
   - There must be at least one character between the FN characters and the ending dollar sign or percent character. FN$ and FN% are not valid function names.

7. *Unsubs-vbl-nam* specifies optional formal function parameters.

8. You can specify the *data-type* of function parameters with a data-type keyword. If you do not specify a *data-type*, parameters are of the default type and size. Parameters that follow a data-type keyword are of the specified type and size until you specify another *data-type*.

9. *BASIC–PLUS–2* allows you to specify up to eight parameters in a DEF* statement.

10. *VAX–11 BASIC* allows you to specify up to 255 parameters in a DEF* statement.

*Single-Line DEF**

1. *Exp* specifies the operations the function performs.

*Multi-Line DEF**

1. *Statements* specify the operations the function performs.

2. The END DEF or FNEND statement is required to end a multi-line DEF*.

3. You can use any BASIC statement except END FUNCTION, END SUB, FUNCTION, FUNCTIONEND, FUNCTIONEXIT, DEF, or DEF* in a function definition.

General Rules

1. When BASIC encounters a DEF* statement, control of the program passes to the next executable statement after the DEF.
2. Functions are invoked when you use the function name in an expression.
3. You cannot specify how parameters are passed. When you invoke a DEF* function, BASIC evaluates parameters from left to right and passes parameters to the function so that they cannot be modified. Numeric parameters are passed BY VALUE, and string parameters are passed BY DESC, where the descriptor points to a local copy. DEF* functions may reference variables in the main program, but they cannot reference variables in other DEF or DEF* functions. A DEF* function may, therefore, modify variables in the program, but not variables within another DEF* function.
4. A DEF* is local to the program or subprogram that defines it.
5. The DEF* statement permits inclusion of the GOTO, ON GOTO, GOSUB, and ON GOSUB statements in a multi–line DEF* function. This allows you to transfer program control outside the function definition.
6. Although other variables used within the body of a DEF* are not local to the DEF*, DEF* formal parameters are. However, if you change the value of formal parameters within a DEF* function and then transfer control out of the DEF* without executing the END DEF or FNEND statement, variables outside the DEF* that have the same names as DEF* formal parameters are also changed.
7. The DEF* statement, or the first invocation of a function, whichever occurs first, constitutes the declaration of the function. The DECLARE FUNCTION statement defines the name of the function, but does not invoke it.
8. If your program invokes a function before the DEF* statement defines the function, or if the number of parameters, types of parameters, or type of result declared in the invocation disagree with the number or types of parameters defined in the DEF* statement, BASIC signals an error.
9. DEF* function values are not initialized when DEF* functions are invoked. Therefore, if a DEF* is invoked, and no new function value is assigned, the DEF* returns the value of its previous invocation.
10. DEF* functions can be recursive.
11. DATA statements in a multi-line DEF* are not local to the function; they are local to the program module containing the function definition.
12. The error handler of the program module that contains the DEF* is the default error handler for a DEF* function, not ON ERROR GO BACK as in DEF functions. Parameters return to their original values when control passes to the error handler.

Examples

*Single-Line DEF**

```
1000     DEF* STRING CONCAT(STRING A,B) = A + B
2000     INPUT 'Enter two words';WORD1,WORD2
2010     PRINT CONCAT(WORD1,WORD2)
```

*Multi-Line DEF**

```
1000     DEF* DOUBLE EXAMPLE(DOUBLE A, B, SINGLE C, D, E)
               EXIT DEF IF B = 0
               EXAMPLE = (A/B) + C - (DE)
1030     END DEF
1040     INPUT 'Enter 5 numbers';V,W,X,Y,Z
1050     PRINT EXAMPLE(V,W,X,Y,Z)
```

10.0 DELETE

Function

The DELETE statement removes a record from a relative or indexed file.

Format

```
DELETE chnl-exp
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

General Rules

1. The DELETE statement removes the current record from a file. You cannot then access the record.
2. The file specified by *chnl-exp* must be open with ACCESS MODIFY or WRITE.
3. You can delete a record only if the last I/O statement executed on the specified channel was a successful GET or FIND.
4. The DELETE statement leaves the Current Record Pointer undefined and the Next Record Pointer unchanged.
5. BASIC signals an error when the I/O channel is illegal or not open, when no current record exists, when access is illegal or illogical, when the operation is illegal, or when the record or bucket is locked.
6. In *VAX–11 BASIC*, if the record being deleted is in a file opened with UNLOCK EXPLICIT, the DELETE statement does not remove the lock on the record. If no lock was imposed with a previous GET or FIND statement, the default lock, ALLOW NONE, remains imposed. The lock can be removed with the FREE or UNLOCK statement. See the sections on GET, FIND, OPEN, FREE, and UNLOCK in this manual for more information on explicit record locking and unlocking.

VAX

Examples

```
1000    DELETE 5
```

11.0 DIMENSION

Function

The DIMENSION statement creates and names a static, dynamic, or virtual array. The array subscripts determine the dimensions and size of the array. You can specify the data type of the array and associate the array with an I/O channel.

Format

Nonvirtual, Nonexecutable

```
{ DIM       }
{ DIMENSION }   { [ data-type ] array-nam ( int-const,... ) },...
```

Virtual

```
{ DIM       }
{ DIMENSION }   chnl-exp, { [ data-type ] array-nam ( int-const,... ) [ = int-const ] },...
```

Executable

```
{ DIM       }
{ DIMENSION }   { [ data-type ] array-nam ( int-vbl,... ) },...
```

Syntax Rules

1. *Array-nam* is an array name. It must conform to the rules for naming variables.

2. An *array-nam* in a DIM statement cannot also appear in a COMMON, MAP, or DECLARE statement.

3. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.

4. If you do not specify a data type, the *array-nam* determines the type of data the array holds. If the *array-nam* ends in a percent sign (%), the array stores integer data. If the *array-nam* ends in a dollar sign ($), the array stores string data. Otherwise, the array stores data of the default type and size.

5. A *VAX–11 BASIC* array can have up to 32 dimensions.

6. A *BASIC–PLUS–2* array can have up to eight dimensions.

7. Each instance of *int-const* or *int-vbl* within the parentheses specifies the upper bound of an array dimension.

   - In *VAX–11 BASIC*, array bounds must be in the range 0 to 2^{31-1}.
   - In *BASIC–PLUS–2*, array bounds must be in the range 0 to 2^{15-1}.
   - Although the compiler does not generate an error for subscript values outside of these ranges, there is a limit to the amount of storage your system can allocate. Therefore, very large arrays can cause an internal allocation error or a run-time error.

Nonvirtual, Nonexecutable

1. When all the dimension specifications are *int-consts*, as in DIM A(15%,10%,20%), the DIM statement is nonexecutable and the array is static. A static array cannot appear in another DIM statement because BASIC allocates storage at compile time.

2. A nonexecutable DIM statement must lexically precede any reference to the array it dimensions. That is, you must DIMENSION a static array before you can reference array elements.

Virtual

1. The pound sign (#) must precede *chnl-exp* when dimensioning virtual arrays.

2. The virtual array must be dimensioned and the file must be open before you can reference the array.

3. When the *data-type* is STRING, the =*int-const* clause specifies the length of each array element. The default string length is 16 characters. Virtual string array lengths are rounded to the next higher power of two.

Executable

1. When any of the dimension specifications are *int-vbl*s, as in DIM A(10%,20%,Y%), the DIM statement is executable and the array is dynamic. A dynamic array can be redimensioned with a DIM statement any number of times, since BASIC allocates storage at run time.

General Rules

1. You can create an array implicitly by referencing an array element without using a DIM statement. This causes BASIC to create an array with dimensions of (10), (10,10), (10,10,10), and so on, depending on the number of bounds specifications in the referenced array element. You cannot create virtual or executable arrays implicitly.

2. The lower bound of a BASIC array is always zero, rather than one. Thus, A(10) allocates 11 elements, A(10,10) allocates 121 elements, and A(0,0,0) allocates 1 element.

3. BASIC allocates storage for arrays by row, from right to left.

Nonvirtual, Nonexecutable

1. You can declare arrays with the COMMON, MAP, and DECLARE statements. Arrays so declared cannot be redimensioned with the DIM statement. Furthermore, string arrays declared with a COMMON or MAP statement are always fixed-length.

2. If you reference an array element declared in an array whose subscripts are larger than the bounds specified in the DIM statement, BASIC signals the error "Subscript out of range" (ERR = 55).

Virtual

1. When the rightmost subscript varies faster than the subscripts to the left, fewer disk accesses are necessary to access array elements in virtual arrays.

2. Using the same DIM statement for multiple virtual arrays allocates all arrays in a single disk file. The arrays are stored in the order they were declared.

3. Any program or subprogram can access a virtual array by declaring it in a *virtual* DIMENSION statement. For example:

```
100     DIM #1, A(10)
200     DIM #1, B(10)
```

   In this example, array B overlays array A. You must, however, specify the same channel number, data types, and limits in the same order as they occur in the DIM statement that created the virtual array.

4. BASIC stores a string in a virtual array by padding it with trailing nulls to the length of the array element. It removes these nulls when it retrieves the string from the virtual array.

5. In *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* and in *VAX–11 BASIC*, the OPEN statement for a virtual array must include the ORGANIZATION VIRTUAL clause for the *chnl-exp* specified in the DIMENSION statement.

6. BASIC does not initialize virtual arrays and treats them as statically allocated arrays. You cannot redimension virtual arrays.

7. Refer to the *BASIC User's Guide* for more information on virtual arrays.

Executable

1. You create an executable, dynamic array by using integer variables for array bounds as in DIM A(Y%,X%). This eliminates the need to dimension an array to its largest possible size. Array bounds in an executable DIM statement can be constants or variables, but not expressions. At least one bound must be a variable.

2. You cannot reference an array named in an executable DIM statement until after the DIM statement executes.

3. You can redimension a dynamic array to make the bounds of each dimension larger or smaller, but you cannot change the number of dimensions. That is, you cannot redimension a four-dimensional array to be a five-dimensional array.

4. The executable DIM statement cannot be used to dimension virtual arrays, arrays received as formal parameters, or arrays declared in COMMON, MAP, or nonexecutable DIM statements.
5. An executable DIM statement always reinitializes the array to zero (for numeric arrays) or the null string (for string arrays).
6. If you reference an array element declared in an executable DIM statement whose subscripts are larger than the bounds specified in the last execution of the DIM, BASIC signals the error "Subscript out of range" (ERR=55).

Examples

Nonvirtual, Nonexecutable

```
300      DIM STRING NAME_LIST(100,100), BYTE AGE(100)
```

Virtual

```
100      DIM #1%, STRING NAM_LIST(500), REAL AMOUNT(10,10)
```

Executable

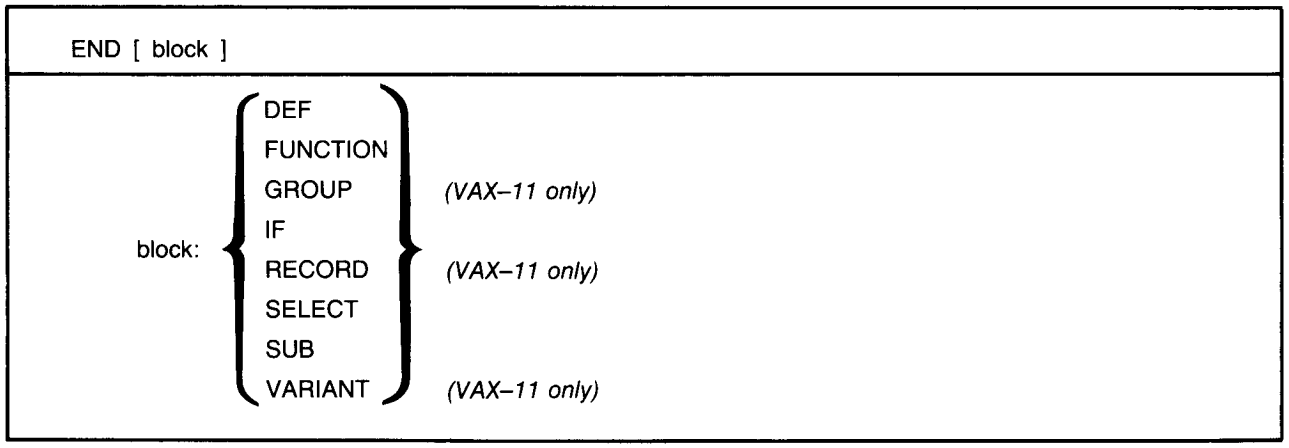
```
200      DIM DOUBLE INVENTORY(BASE,MARKUP)
```

12.0 END

Function

The END statement marks the physical and logical end of a main program, a program module, or a block of statements.

Format



Syntax Rules

1. The END statement with no *block* keyword marks the end of a main program. The END statement must be the last statement on the lexically last line in the main program.
2. The END statement followed by a *block* keyword marks the end of a BASIC SUB or FUNCTION subprogram, or a DEF, IF, or SELECT statement block. In *VAX–11 BASIC*, END RECORD, END GROUP, and END VARIANT mark the end of a RECORD statement, or a GROUP component or VARIANT component of a RECORD statement.
3. The END *block* statement must be the lexically last statement in a subprogram or statement block and must match the statement that established the subprogram or statement block.

General Rules

1. When an END statement marking the end of a main program executes, BASIC closes all files and releases all program storage.
2. BASIC cannot execute an END statement that marks the end of a program unit while an error is being handled. The module must execute a RESUME or ON ERROR statement before the END statement.
3. BASIC signals an error when a program contains an END *block* statement with no corresponding and preceding *block* keyword.
4. When BASIC executes an END DEF or END FUNCTION statement, it returns the function value to the statement that invoked the function and releases all storage associated with the DEF or FUNCTION.

5. The END DEF statement restores the error handling in effect when the DEF was invoked.
6. Error handlers set up in DEF* statements are global. The END DEF statement does not restore the error handling in effect when the DEF* was invoked.
7. The END SUB and END FUNCTION statements do not affect I/O operations or files.
8. The END SUB statement releases the storage allocated to local variables and returns control to the calling program.
9. The END SUB statement cannot be executed in an error handler unless the SUBEND is in a subprogram called by the error handler.

Examples

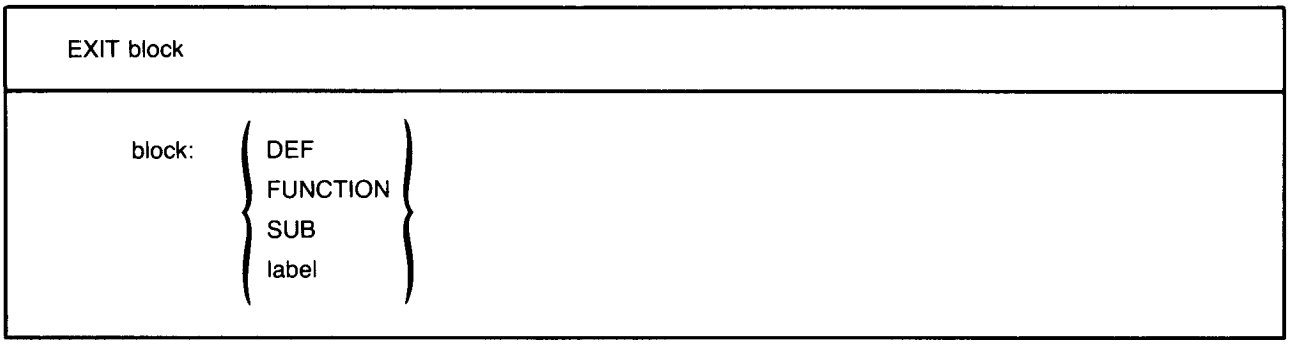
```
300       IF A = 20
          THEN      PRINT "Bye"
                    GOTO 32767
          ELSE GOTO 100
          END IF
          !
          !
          !
32767     END
```

13.0 EXIT

Function

The EXIT statement lets you exit from a SUB or FUNCTION subprogram, a multi-line DEF, or from a statement block.

Format



Syntax Rules

1. The FUNCTION, SUB, and DEF keywords specify the type of subprogram or multi-line DEF from which BASIC is to exit.

2. *Label* specifies a statement label for an IF, SELECT, FOR, WHILE, or UNTIL statement block.

General Rules

1. An EXIT DEF, EXIT FUNCTION, or EXIT SUB statement is equivalent to an unconditional branch to an END DEF, END FUNCTION, or END SUB statement. Control then passes to the statement that invoked the DEF or to the statement following the statement that called the subprogram.

2. The EXIT *label* statement is equivalent to an unconditional branch to the first statement following the end of the IF, SELECT, FOR, WHILE, or UNTIL statement labelled by the *label*.

3. An error handler cannot execute an EXIT FUNCTION or EXIT SUB statement unless the error handler calls the FUNCTION or SUB subprogram.

4. An EXIT FUNCTION or EXIT SUB statement cannot be used within a multi-line DEF function.

5. When the EXIT FUNCTION or EXIT SUB statement executes, BASIC releases all storage allocated to local variables and returns control to the calling program.

Examples

```
100      LOOP_1: FOR I% = 1% TO 10%
                     PRINT I%
                     IF I% = 5%
                     THEN EXIT LOOP_1
                     END IF
               NEXT I%
         !
         !
         !
5000     SUB SUBA
         !
         !
         !
6000     EXIT SUB
         !
         !
         !
10000    END SUB
```

14.0 EXTERNAL

Function

The EXTERNAL statement declares constants, variables, functions, and subroutines external to your program. You can describe parameters for external functions and subroutines.

Format

External Constants

```
EXTERNAL data-type CONSTANT const-nam,...
```

External Variables

```
EXTERNAL data-type unsubs-vbl-nam,...
```

External Functions

```
EXTERNAL data-type FUNCTION { func-nam [ pass-mech ] [ ( [ external-param ],... ) ] },...
```

External Subroutines

```
EXTERNAL SUB { sub-nam [ pass-mech ] [ ( [ external-param ],... ) ] },...
```

```
pass-mech:      { BY DESC
                { BY REF
                { BY VALUE

external-param:   [ data-type ] [ DIM ([,]...) ] [ = int-const ] [ pass-mech ]
```

Syntax Rules

1. For external variables, *data-type* can be any valid numeric data type.
2. For external constants, *data-type* can be:
   - For *VAX–11 BASIC*: BYTE, WORD, LONG, SINGLE, INTEGER (any size), or REAL (if the default size is SINGLE).
   - For *BASIC–PLUS–2*: WORD, or INTEGER (if the default size is WORD).
3. For external functions and subroutines, *data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined by a RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.

4. In *VAX–11 BASIC*, the name of an external constant, variable, function, or subroutine can consist of from 1 to 31 characters and must conform to the following rules: VAX

   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), periods (.), and underscores (_).
   - Quoted names are allowed for the EXTERNAL SUB statement only. Quoted names can consist of any combination of printable ASCII characters.
   - An EXTERNAL SUB or EXTERNAL FUNCTION statement with empty parentheses specifies that the named subprogram has zero arguments.

5. An EXTERNAL SUB or EXTERNAL FUNCTION statement with no parentheses specifies that the named subprogram may receive any number of arguments.

6. In *BASIC–PLUS–2*, the name of an external constant, variable, or subroutine can consist of from one to six characters and must conform to the following rules: BP2

   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), and periods (.).
   - Quoted names are allowed for the EXTERNAL SUB statement only. Quoted names can consist of any combination of alphabetic characters, digits, dollar signs, periods, and spaces.

External Functions and Subroutines

1. The *data-type* that precedes the FUNCTION keyword defines the data type of the function result.
2. *Pass-mech* specifies how parameters are to be passed to the function or subroutine.
   - A *pass-mech* clause outside the parentheses applies to all parameters.
   - A *pass-mech* clause inside the parentheses overrides the previous *pass-mech* and applies only to the specific parameter.
3. *External-param* defines the form of the arguments passed to the external function or subprogram.
   - Empty parentheses indicate that the function or subroutine is being named, but that parameters are not being defined.
   - *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type.

- The DIM keyword indicates that the parameter is an array. Commas specify array dimensions. The number of dimensions is equal to the number of commas plus one. For example:

```
100     EXTERNAL STRING FUNCTION NEW (DOUBLE, STRING DIM(,), DIM())
```

  This example declares a function named NEW that has three parameters. The first is a double-precision floating-point value, the second is a two-dimensional string array, and the third is a one-dimensional string array. The function returns a string result.

- You can specify how an argument is to be passed to the function or subprogram with the optional *pass-mech* clause. If you do not specify a passing mechanism for a parameter, BASIC passes arguments by the default passing mechanisms listed in Tables 19 and 20.

General Rules

1. The EXTERNAL statement must precede any program reference to the constant, variable, function, or subroutine declared in the statement.
2. The EXTERNAL statement is not executable.
3. A name declared in an EXTERNAL CONSTANT statement may be used in any nondeclarative statement as if it were a constant.
4. A name declared in an EXTERNAL FUNCTION statement may be used as a function invocation in an expression.
5. A name declared in an EXTERNAL SUB statement may be used in a CALL statement.
6. The optional *pass-mech* clauses in the EXTERNAL FUNCTION and EXTERNAL SUB statements tell BASIC how to pass arguments to a non–BASIC function or subprogram. Table 19 describes *VAX–11 BASIC* parameter passing mechanisms. Table 20 describes *BASIC–PLUS–2* parameter passing mechanisms.

   - BY REF specifies that BASIC passes the argument's address. This is the default for all arguments except strings and entire arrays.
   - BY VALUE specifies that *VAX–11 BASIC* passes the argument's 32–bit value and that *BASIC–PLUS–2* passes the argument's 16–bit value.
   - BY DESC specifies that BASIC passes the address of a *VAX–11 BASIC* descriptor or a *BASIC–PLUS–2* descriptor. For information about the format of a *VAX–11 BASIC* descriptor for strings and arrays, see Appendix C in *BASIC on VAX/VMS Systems*. *BASIC–PLUS–2* creates descriptors only for strings and arrays; these descriptors are described in Appendix C in *BASIC on RSX–11M/M–PLUS Systems* and *BASIC on RSTS/E Systems*.

7. The arguments passed to external functions and subroutines should match the external parameters declared in the EXTERNAL FUNCTION or EXTERNAL SUB statement in number, type, ordinality, and passing mechanism as BASIC forces arguments to conform to the declared parameters. BASIC signals an error when conformance is impossible (for example, when a STRING argument is passed where an INTEGER parameter was declared) and an informational message when a conversion results in a modifiable parameter becoming a nonmodifiable parameter.

Examples

External Constants

```
100      EXTERNAL LONG CONSTANT SYS$FC
```

External Variables

```
100      EXTERNAL WORD SYSNUM
```

External Functions

```
100      EXTERNAL DOUBLE FUNCTION USR$2 (DOUBLE DIM(,),BYTE BY VALUE)
```

External Subroutines

```
100      EXTERNAL SUB CALC BY DESC (STRING DIM(,), BYTE BY REF)
```

15.0 FIELD

Function

Note

The FIELD statement is supported only for compatibility with BASIC–PLUS. Because data defined in the FIELD statement can be accessed only as string data, you must use the CVTxx functions to process numeric data. This means that you must convert string data to numeric after you move it from the I/O buffer. Then, after processing, you must convert numeric data back to string data before transferring it to the I/O buffer. DIGITAL recommends that you use BASIC's dynamic mapping feature or multiple MAPs instead of the FIELD statement and CVTxx functions.

The FIELD statement dynamically associates string variables with all or parts of an I/O buffer. FIELD statements do not move data. Instead, they permit direct access through string variables to sections of a specified I/O buffer.

Format

```
FIELD chnl-exp, int-exp AS str-vbl [ , int-exp AS str-vbl ]...
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be preceded by a pound sign (#). A file must be open on the specified channel or BASIC signals an error.

2. *Int-exp* specifies the number of characters in the *str-vbl* that follows the AS keyword.

General Rules

1. A FIELD statement is executable. You can change a buffer description at any time by executing another FIELD statement. For example:

```
100     FIELD #1%, 40% AS WHOLE_FIELD$
        FIELD #1%, 10% AS A$, 10% AS B$, 10% AS C$, 10% AS D$
```

   The first FIELD statement associates the first 40 characters of a buffer with the variable WHOLE_FIELD$. The second FIELD statement associates the first 10 characters of the same buffer with A$, the second 10 characters with B$, and so on. Later program statements can refer to any of the variables named in the FIELD statements to access specific portions of the buffer.

2. You cannot define virtual array strings as string variables in a FIELD statement.

3. See the *BASIC–PLUS Language Manual* for more information on the FIELD statement.

VAX–11 BASIC

1. A variable named in a FIELD statement cannot be used in a COMMON or MAP statement, as a parameter in a CALL or SUB statement, or in a MOVE statement.
2. Using the FIELD statement on a VIRTUAL file that contains a virtual array causes BASIC to signal "Illegal or illogical access" (ERR=136).
3. If you name an array in a FIELD statement, you cannot use MAT statements of the format:

   MAT arr-nam1 = arr-nam2

   or

   MAT arr-nam1 = NUL$

   where *arr-nam1* is named in the FIELD statement. An attempt to do so causes BASIC to signal a compile-time error.

Examples

```
100     FIELD #8%, 2% AS U$, 2% AS CL$, 4% AS X$, 4% AS Y$
210     LSET U$ = CVT%$(U%)
        LSET CL$ = CVT%$(CL%)
        LSET X$ = CVTF$(X)
        LSET Y$ = CVTF$(Y)
300     U% = CVT$%(U$)
        CL% = CVT$%(CL$)
        X = CVT$F(X$)
        Y = CVT$F(Y$)
```

Note

DIGITAL does not recommend the FIELD statement for new program development.

16.0 FIND

Function

The FIND statement locates a specified record in a disk file and makes it the Current Record for a GET, UPDATE, or DELETE operation. FIND statements are valid on RMS sequential, relative, indexed, and block I/O files. You should not use FIND statements on terminal-format files, virtual array files, or files opened with ORGANIZATION UNDEFINED.

Format

VAX–11 BASIC


```
FIND chnl-exp [ , position-clause ] [ , lock-clause ]

position-clause:  { RFA rfa-exp
                  { RECORD num-exp
                  { KEY# key-clause

lock-clause:      { ALLOW allow-clause
                  { REGARDLESS

allow-clause:     { NONE
                  { READ
                  { MODIFY

key-clause:       int-exp1 rel-op { str-exp
                                  { int-exp2
                                  { decimal-exp

rel-op:           { EQ
                  { GE
                  { GT
```

(continued on next page)

FIND

BP2 *BASIC–PLUS–2*

```
FIND chnl-exp [ , position-clause ]

position-clause:  { RFA rfa-exp          }
                  { RECORD num-exp       }
                  { KEY# key-clause      }

key-clause:       int-exp1 rel-op { str-exp  }
                                  { int-exp2 }

rel-op:           { EQ }
                  { GE }
                  { GT }
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

2. *Position-clause* specifies the position of a record in a file. BASIC signals an error if you specify a *position-clause* and *chnl-exp* is not associated with a disk file.

   - If you do not specify a *position-clause*, FIND locates records sequentially. Sequential record access is valid on RMS sequential, relative, indexed, and block I/O files.

   - The RFA *position-clause* allows you to randomly locate records by specifying the Record File Address (RFA) of a record. That is, you specify the disk address of a record, and RMS locates the record at that address. All RMS file organizations may be accessed by RFA.

   - The RECORD *position-clause* allows you to randomly locate records in relative and block I/O files by specifying the record number.

   - The KEY *position-clause* allows you to randomly locate records in indexed files by specifying a key of reference, a relational test, and a key value.

3. *Rfa-exp* in the RFA *position-clause* is a variable of the RFA data type that specifies the record's Record File Address. Note that an RFA expression can only be a variable of the RFA data type or the GETRFA function. Use the GETRFA function to find the RFA of a record.

4. *Int-exp* in the RECORD *position-clause* specifies the number of the record you want to locate. It must be between one and the file's maximum record number.

5. In the *key-clause*:

   - *Int-exp1* is the target key of reference. It must be a WORD or LONG integer between zero and the highest-numbered key for the file, inclusive. BASIC converts BYTE integers to WORD. The primary key is key number zero, the first alternate key is key number one, the second alternate key is key number two, and so on. *Int-exp1* must be preceded by a pound sign (#) or BASIC signals an error.

   - *Str-exp* and *int-exp2* specify a string or integer value to be compared with the key value of a record.

   - *Rel-op* specifies how *str-exp* or *int-exp2* is to be compared to *int-exp1*. EQ means "equal to," GE means "greater than or equal to," and GT means "greater than."

6. When you specify a RECORD clause, *chnl-exp* must be a channel associated with an open relative or block I/O file.

7. When you specify a KEY clause, *chnl-exp* must be a channel associated with an open indexed file.

VAX–11 BASIC

1. *Str-exp* in the KEY clause cannot be a null string.

2. *Decimal-exp* in the KEY clause specifies a packed decimal value to be compared with the key value of a record.

3. *Lock-clause* allows you to control how a record is locked to other access streams. The file associated with *chnl-exp* must have been opened with the UNLOCK EXPLICIT clause or BASIC signals the error "illegal record locking clause".

4. If you specify a *lock-clause*, it must follow the *position-clause*. If the *lock-clause* precedes the *position-clause*, BASIC signals an error.

General Rules

1. The file associated with *chnl-exp* must be opened with ACCESS MODIFY, READ, or SCRATCH before your program can execute a FIND.

2. FIND does not transfer any data.

3. A successful sequential FIND updates both the Current Record and Next Record Pointers.

   - For sequential files, a successful FIND locates the next sequential record (the record pointed to by the Next Record Pointer) in the file, changes the Current Record Pointer to the record just found, and sets the Next Record Pointer to the next sequential record. If the Current Record Pointer points to the last record in a file, a sequential find causes BASIC to signal "End of file on device" (ERR=11).

   - For relative files, a successful FIND locates the record with the next higher record number (or cell number), makes it the Current Record, and changes the Next Record to the Current Record plus one.

- For indexed files, a successful FIND locates the next logical record in the current key of reference, makes this the Current Record, and changes the Next Record to the Current Record plus one.
- For block I/O files, a successful FIND locates the next disk block (for files with RECORDSIZE 512) or the next record (for files with RECORDSIZE greater than 512), makes it the Current Record, and changes the Next Record to the Current Record plus one.

4. A successful random FIND by KEY locates the first record whose key satisfies the *key-clause* comparison:
   - With an exact key match (EQ), a successful FIND locates the first record in the file that equals the key value given in *int-exp* or specified by *str-exp*. The characters specified by *str-exp* are matched approximately rather than exactly. That is, if you specify "ABC" and the key length is six characters, BASIC matches the first record that begins with ABC. If you specify "ABC ", BASIC matches only a record with the key "ABC ". If no match is possible, BASIC signals the error "Record not found" (ERR=155).
   - With the greater than key match (GT), a successful FIND locates the first record with a value greater than *int-exp* or *str-exp*. If no such record exists, BASIC signals the error "End of file on device" (ERR=11).
   - If you specify a greater than or equal to key match (GE), a successful FIND locates the first record that equals the key value in *int-exp* or *str-exp*. If no exact match is possible, BASIC locates the first record with a key value higher than *int-exp* or *str-exp*.
5. A successful random access FIND by RFA or by RECORD changes the Current Record Pointer to the record specified by *rfa-exp* or *int-exp*, but leaves the Next Record Pointer unchanged.
6. A successful random access FIND by KEY changes the Current Record Pointer to the first record whose key satisfies the *key-clause* comparison and the Next Record Pointer to the record with the next higher value in the current key.
7. When a random access FIND by RFA, RECORD, or KEY is not successful, BASIC signals "Record not found" (ERR=155). The values of the Current Record Pointer and Next Record Pointer are undefined.
8. If the RMS index lists are in memory, a FIND on an indexed file does not initiate any disk operations.

VAX *VAX–11 BASIC*

1. The type of lock you impose on a record remains in effect until you explicitly unlock it with a FREE or UNLOCK statement or until you close the file.
   - ALLOW NONE specifies no access to the record. This means that other access streams cannot retrieve the record unless they bypass lock checking with GET REGARDLESS.
   - ALLOW READ specifies read access to the record. This means that other access streams can retrieve the record but cannot PUT or UPDATE the record.
   - ALLOW MODIFY specifies both read and write access to the record. This means that other access streams can GET, PUT, DELETE, or UPDATE the record.

2. When you do not specify an ALLOW clause, locking is imposed as follows:

   - If the file associated with *chnl-exp* was opened with UNLOCK EXPLICIT, BASIC imposes the ALLOW NONE lock on the retrieved record and the next GET or FIND does not unlock the previously locked record.
   - If the file associated with *chnl-exp* was not opened with UNLOCK EXPLICIT, BASIC locks the retrieved record and unlocks the previously locked record.

BASIC–PLUS–2 (BP2)

1. When you access a shared file, a successful FIND locks the record or bucket and unlocks the previously locked record or bucket.

Examples

Sequential Access

```
100     MAP (XYZ) STRING LAST_NAME = 10, FIRST_NAME = 6
150     OPEN 'EMP.DAT' AS FILE #1, &
            ORGANIZATION SEQUENTIAL, MAP XYZ
200     FIND #1
```

Random Access

```
100     DECLARE RFA Address(99)
200     MAP (XYZ) STRING LAST_NAME = 10, FIRST_NAME = 6
300     OPEN 'EMP.DAT' AS FILE #1, &
            ORGANIZATION SEQUENTIAL, MAP XYZ
400     FIND #1
        Address(0) = GETRFA(1)
        !
        !
        !
500     FIND #1, RFA Address(5)
        !
        !
        !
600     OPEN 'NEWEMP.DAT' AS FILE #2, &
            ORGANIZATION RELATIVE, MAP XYZ
700     FIND #2, RECORD A%
        Address(A%) = GETRFA(2)
        !
        !
        !
        FIND #2, RFA Address(A%)
        !
        !
        !
900     OPEN 'OLDEMP.DAT' AS FILE #3, &
            ORGANIZATION INDEXED, MAP XYZ, &
            PRIMARY KEY LAST_NAME
        FIND #3, KEY #0 EQ "JONES"
        Address(5) = GETRFA(3)
        !
        !
        !
        FIND #3, RFA Address(7)
```

VAX–11 BASIC

```
100     MAP (XYZ) STRING LAST_NAME = 10, FIRST_NAME = 6
200     OPEN 'EMP.DAT' AS FILE #1,                          &
             ORGANIZATION INDEXED,                          &
             MAP XYZ, PRIMARY KEY LAST_NAME,                &
             UNLOCK EXPLICIT
400     FIND #3, KEY #0 EQ "JONES", ALLOW READ
```

17.0 FNEND

Function

The FNEND statement is a synonym for END DEF. See the END statement for syntax rules.

Format

```
{ FNEND   }
{ END DEF }
```

18.0 FNEXIT

Function

The FNEXIT statement is a synonym for the EXIT DEF statement. See the EXIT statement for syntax rules.

Format

19.0 FOR

Function

The FOR statement repeatedly executes a block of statements, while incrementing a specified control variable for each execution of the statement block. FOR loops can be conditional or unconditional, and can modify other statements.

Format

Unconditional

```
FOR num-unsubs-vbl = num-exp1 TO num-exp2 [ STEP num-exp3 ]
        [ statement ]...
NEXT num-unsubs-vbl
```

Conditional

```
                                                  { UNTIL }
FOR num-unsubs-vbl = num-exp1 [ STEP num-exp3 ]   { WHILE }   cond-exp
        [ statement ]...
NEXT num-unsubs-vbl
```

Unconditional Statement Modifier

```
statement FOR num-unsubs-vbl = num-exp1 TO num-exp2 [ STEP num-exp3 ]
```

Conditional Statement Modifier

```
                                                        { UNTIL }
statement FOR num-unsubs-vbl = num-exp1 [ step num-exp3 ] { WHILE }   cond-exp
```

Syntax Rules

1. *Num-unsubs-vbl* is the loop variable. It is incremented each time the loop executes.
2. In unconditional FOR loops, *num-exp1* is the initial value of the loop variable, while *num-exp2* is the maximum value.
3. In conditional FOR loops, *num-exp1* is the initial value of the loop variable, while the *cond-exp* in the WHILE or UNTIL clause is the condition that controls loop iteration.
4. *Num-exp3* in the STEP clause is the value by which the loop variable is incremented after each execution of the loop.
5. In *VAX–11 BASIC*, you can nest FOR loops to a maximum of 12 levels, depending on the complexity of the loops.

6. In *BASIC–PLUS–2*, you nest FOR loops to a maximum of 8 levels, depending on the complexity of the loops.
7. An inner loop must be entirely within an outer loop; the loops cannot overlap.
8. You cannot use the same loop variable in nested FOR loops. That is, if the outer loop uses "FOR I = 1 TO 10", you cannot use the variable I as a loop variable in an inner loop.
9. The default for *num-exp3* is one if there is no STEP clause.
10. You can transfer control into a FOR loop only be returning from a function invocation, a subprogram call, or an error handler that was invoked in the loop.
11. Each FOR statement must have a corresponding NEXT statement or BASIC signals and error.

General Rules

1. The starting, incrementing, and ending values of the loop do not change during loop execution.
2. The loop variable can be modified inside the FOR loop.
3. BASIC converts *num-exp1*, *num-exp2*, and *num-exp3* to the data type of *num-unsubs-vbl* (the loop variable) before storing them.
4. When an unconditional FOR loop ends, the loop variable contains the value last used in the loop, not the value that caused loop termination.
5. During each iteration of a conditional loop, BASIC tests the value of *cond-exp* before it executes the loop.
   - If you specify a WHILE clause and *cond-exp* is false (value zero), BASIC exits from the loop. If the *cond-exp* is true (value nonzero), the loop executes again.
   - If you specify an UNTIL clause and *cond-exp* is true (value nonzero), BASIC exits from the loop. If the *exp* is false (value zero), the loop executes again.
6. When FOR is used as a statement modifier, BASIC executes the statement until *num-unsubs-vbl* equals or exceeds *num-exp2* or until the WHILE or UNLESS condition is satisfied.

Examples

Unconditional

```
350     FOR I = 3 TO 99 STEP 3
        !
        !
        !
400     NEXT I
```

Unconditional

```
100     FOR Z = 0 STEP 2 UNTIL X
        !
        !
        !
200     NEXT Z
```

Unconditional Statement Modifier

```
100      A = A + .0005 FOR I = 1 TO 10
```

Conditional Statement Modifier

```
100      FIND #2 FOR I = 1 UNTIL ERR=155
```

20.0 FREE (VAX–11 BASIC)

Function

The FREE statement unlocks all records and buckets associated with a specified channel.

Format

```
FREE chnl-exp
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

General Rules

1. The file specified by *chnl-exp* must be open.
2. You cannot use the FREE statement with files not on disk.
3. If there are no locked records or buckets on the specified channel, the FREE statement has no effect and BASIC does not signal an error.
4. The FREE statement does not change record buffers or pointers.
5. Your program must execute a GET or FIND statement after a FREE statement executes before a PUT statement can execute.

Examples

```
450       FREE #6%
```

21.0 FUNCTION

Function

The FUNCTION statement marks the beginning of a FUNCTION subprogram and defines the subprogram's parameters.

Format

VAX–11 BASIC

VAX

```
FUNCTION data-type func-nam [ pass-mech ] [ ( [ formal-param ],... ) ]
        [ statement ]...
{ END FUNCTION  }
{ FUNCTIONEND   }

pass-mech:      { BY REF  }
                { BY DESC }

formal-param:   [ data-type ] { unsubs-vbl-nam                                    }
                              { array-nam ( [int-const] ,... ) } [ = int-const ] [ pass-mech ]
                              {             [,        ] ...    }
```

BASIC–PLUS–2

BP2

```
FUNCTION data-type func-nam [ ( [ formal-param ],... ) ]
        [ statement ]...
{ END FUNCTION  }
{ FUNCTIONEND   }

formal-param:   [ data-type ] { unsubs-vbl-nam                                    }
                              { array-nam ( [int-const] ,... ) }
                              {             [,        ] ...    }
```

Syntax Rules

1. *Func-nam* names the FUNCTION subprogram. The last character of the name cannot be a dollar sign ($).
2. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.
3. The *data-type* that precedes the *func-nam* specifies the data type of the value returned by the FUNCTION subprogram.
4. *Formal-param* specifies the number and type of parameters for the arguments the FUNCTION subprogram expects to receive when invoked.
   - Empty parentheses indicate that the FUNCTION subprogram has zero parameters.
   - *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type.
   - If you specify a *data-type*, *unsubs-vbl-nam* and *array-nam* cannot end in a percent sign (%) or dollar sign ($).
   - Parameters defined in *formal-param* must agree in number and type with the arguments specified in the function invocation.
5. The FUNCTION statement must be the first statement in the FUNCTION subprogram.
6. Compiler directives and comment fields (!), because they are not BASIC statements, may precede the FUNCTION statement. However, they cannot precede the subprogram's first numbered line. Note that REM is a BASIC statement; therefore, it cannot precede the FUNCTION statement.
7. Every FUNCTION statement must have a corresponding END FUNCTION statement or FUNCTIONEND statement.
8. Any BASIC statement except END, SUB, SUBEND, END SUB, or SUBEXIT can appear in a FUNCTION subprogram.

VAX–11 BASIC

1. *Func-nam* can consist of from 1 to 31 characters The first character must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), periods (.), or underscores (_).
2. If the data type is STRING, the *=int-const* clause allows you to specify the length of the string. The default string length is 16.
3. *VAX–11 BASIC* allows you to specify from 1 to 32 *formal-params*.

4. *Pass-mech* specifies the parameter passing mechanism by which the FUNCTION subprogram receives arguments when invoked. A *pass-mech* should be specified only when the FUNCTION subprogram is being called by a non–BASIC program.

5. A *pass-mech* clause outside the parentheses applies by default to all FUNCTION parameters. A *pass-mech* clause in the *formal-param* list overrides the specified default and applies only to the immediately preceding parameter.

BASIC–PLUS–2

1. *Func-nam* can consist of from one to six characters. The first character must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), and periods (.).

2. *BASIC–PLUS–2* allows you to specify from one to eight *formal-params*.

General Rules

1. FUNCTION subprograms must be declared with the EXTERNAL statement before your program can invoke them.

2. FUNCTION subprograms receive parameters BY REF or BY DESC.

   - BY REF specifies that the FUNCTION subprogram receives the argument's address.

   - BY DESC specifies that the FUNCTION subprogram receives the address of a *VAX–11 BASIC* descriptor or a *BASIC–PLUS–2* descriptor. For information about the format of a *VAX–11 BASIC* descriptor for strings and arrays, see Appendix C in *BASIC on VAX/VMS Systems*; for information on other types of descriptors, refer to the *VAX Architecture Handbook*. *BASIC–PLUS–2* creates descriptors only for strings and arrays; these descriptors are described in Appendix C in *BASIC on RSX–11M/M–PLUS Systems* and *BASIC on RSTS/E Systems*.

3. All variables and data, except virtual arrays, COMMON areas, and MAP areas in a FUNCTION subprogram, are local to the subprogram.

4. BASIC initializes local numeric variables to zero and local string variables to the null string each time the FUNCTION subprogram is invoked.

5. ON ERROR GO BACK is the default error handler for a FUNCTION subprogram.

BASIC–PLUS–2

1. *BASIC–PLUS–2* receives numeric *unsubs-vbls* BY REF and string *unsubs-vbls* and entire arrays BY DESC.

VAX–11 BASIC

1. By default, *VAX–11 BASIC* FUNCTION subprograms receive numeric *unsubs-vbls* BY REF, and all other parameters BY DESC. You can override these defaults with a BY clause:

   - Any parameter can be received BY DESC.

   - To receive a string parameter BY REF, you must specify the string length.

   - To receive an entire array BY REF, you must specify the array bounds.

Examples

VAX–11 BASIC only

```
100     FUNCTION GFLOAT SIGMA BY DESC   &
                (GFLOAT A(20,20),       &
                 B, HFLOAT C BY REF)
        !
        !
        !
250     END FUNCTION
```

BASIC–PLUS–2 only

```
100     FUNCTION DOUBLE CALC (SINGLE A, B, DOUBLE C(10,50))
        !
        !
        !
250     END FUNCTION
```

22.0 FUNCTIONEND

Function

The FUNCTIONEND statement is a synonym for the END FUNCTION statement. See the END statement for syntax rules.

Format

```
{ FUNCTIONEND   }
{ END FUNCTION  }
```

23.0 FUNCTIONEXIT

Function

The FUNCTIONEXIT statement is a synonym for the EXIT FUNCTION statement. See the EXIT statement for syntax rules.

Format

```
{ FUNCTIONEXIT }
{ EXIT FUNCTION }
```

24.0 GET

Function

The GET statement moves a record from a file to a record buffer and makes the data available for processing. GET statements are valid on RMS sequential, relative, indexed, and block I/O files, and on *RSTS/E* non–RMS block I/O files. You should not use GET statements on terminal-format files, virtual array files, or files opened with ORGANIZATION UNDEFINED.

Format

VAX–11 BASIC

VAX

```
GET chnl-exp [ , position-clause ] [ , lock-clause ]

position-clause:    { RFA rfa-exp
                    { RECORD num-exp
                    { KEY# key-clause

lock-clause:        { ALLOW allow-clause
                    { REGARDLESS

allow-clause:       { NONE
                    { READ
                    { MODIFY

key-clause:           int-exp1 rel-op { str-exp
                                      { int-exp2
                                      { decimal-exp

rel-op:             { EQ
                    { GE
                    { GT
```

(continued on next page)

GET

BP2 *BASIC–PLUS–2*

```
GET chnl-exp [ , position-clause ]

position-clause:   { RFA rfa-exp
                   { RECORD num-exp
                   { KEY# key-clause

key-clause:        int-exp1 rel-op { str-exp
                                   { int-exp2

rel-op:            { EQ
                   { GE
                   { GT
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

2. *Position-clause* specifies the position of a record in a file. BASIC signals an error if you specify a *position-clause* and *chnl-exp* is not associated with a disk file.

   - If you do not specify a *position-clause*, GET retrieves records sequentially. Sequential record access is valid on RMS sequential, relative, indexed, and block I/O files.

   - The RFA *position-clause* allows you to randomly retrieve records by specifying the Record File Address (RFA) of a record. That is, you specify the disk address of a record, and RMS retrieves the record at that address. All RMS file organizations may be accessed by RFA.

   - The RECORD *position-clause* allows you to randomly retrieve records in relative and block I/O files by specifying the record number.

   - The KEY *position-clause* allows you to randomly retrieve records in indexed files by specifying a key of reference, a relational test, and a key value.

3. *Rfa-exp* in the RFA *position-clause* is an expression of the RFA data type that specifies the record's Record File Address. Note that an RFA expression can be only a variable of the RFA data type or the GETRFA function. Use the GETRFA function to find the RFA of a record.

4. *Int-exp* in the RECORD *position-clause* specifies the number of the record you want to retrieve. It must be between one and the file's maximum record number.

5. In the *key-clause*:

   - *Int-exp1* is the target key of reference. It must be a WORD or LONG integer between zero and the highest-numbered key for the file, inclusive. BASIC converts BYTE integers to WORD. The primary key is key number zero, the first alternate key is key number one, the second alternate key is key number two, and so on. *Int-exp1* must be preceded by a pound sign (#) or BASIC signals an error.
   - *Str-exp* and *int-exp2* specify a string or integer value to be compared with the key value of a record. *Str-exp* can contain fewer characters than the key of the record you want to retrieve.
   - *Rel-op* specifies how *str-exp* or *int-exp2* is to be compared to *int-exp1*. EQ means "equal to," GE means "greater than or equal to," and GT means "greater than."

6. When you specify a RECORD clause, *chnl-exp* must be a channel associated with an open relative or block I/O file.

7. When you specify a KEY clause, *chnl-exp* must be a channel associated with an open indexed file.

VAX–11 BASIC

1. *Str-exp* in the KEY clause cannot be a null string.

2. *Decimal-exp* in the KEY clause specifies a packed decimal value to be compared with the key value of a record.

3. *Lock-clause* allows you to control how a record is locked to other access streams or to override lock checking when accessing shared files that may contain locked records.

4. If you specify a *lock-clause*, it must follow the *position-clause*. If the *lock-clause* precedes the *position-clause*, BASIC signals an error.

5. If you specify an *allow-clause*, the file associated with *chnl-exp* must have been opened with the UNLOCK EXPLICIT clause or BASIC signals the error "illegal record locking clause".

General Rules

1. The file specified by *chnl-exp* must be open with ACCESS READ or MODIFY before your program can execute a GET. The default ACCESS clause is MODIFY.

2. If the last I/O operation was a successful FIND, a sequential GET retrieves the Current Record located by the FIND and sets the Next Record Pointer to the Current Record plus one.

3. If the last I/O operation was not a FIND, a sequential GET retrieves the Next Record and sets the Next Record Pointer to the Current Record plus one.

   - For sequential files, a sequential GET retrieves the next record in the file.
   - For relative and block I/O files, a sequential GET retrieves the record with the next higher cell number.
   - For indexed files, a sequential GET retrieves the record with the next higher value in the current key of reference.

4. A successful random GET by RFA or by RECORD retrieves the record specified by *rfa-exp* or *int-exp*.

5. A successful random GET by KEY retrieves the first record whose key satisfies the *key-clause* comparison:

   - With an exact key match (EQ), a successful GET retrieves the first record in the file that equals the key value given in *int-exp* or specified by *str-exp*. The characters specified by *str-exp* are matched approximately rather than exactly. That is, if you specify "ABC" and the key length is six characters, BASIC matches the first record that begins with ABC. If you specify "ABC ", BASIC matches only a record with the key "ABC ". If no match is possible, BASIC signals the error "Record not found" (ERR=155).

   - With the greater than key match (GT), a successful GET retrieves the first record with a value greater than *int-exp* or *str-exp*. If no such record exists, BASIC signals the error "End of file on device" (ERR=11).

   - If you specify a greater than or equal to key match (GE), a successful GET retrieves the first record that equals the key value in *int-exp* or *str-exp*. If no exact match is possible, BASIC retrieves the first record with a key value higher than *int-exp* or *str-exp*.

6. A successful random GET by RFA, RECORD, or KEY sets the value of the Current Record Pointer to the record just read. The Next Record Pointer is set to the Current Record plus one.

7. An unsuccessful GET leaves the record pointers and the I/O buffer in an undefined state.

8. If the retrieved record is smaller than the receiving buffer, BASIC fills the remaining buffer space with nulls.

9. If the retrieved record is larger than the receiving buffer, BASIC truncates the record and signals an error.

10. A successful GET sets the value of the RECOUNT variable to the number of bytes transferred from the file to the record buffer.

11. Because a GET statement on a block I/O file always transfers an integral number of 512–byte disk blocks, your program must perform record blocking and deblocking. See Chapter 9 in the *BASIC User's Guide* for more information.

VAX–11 BASIC

1. The type of lock you impose on a record remains in effect until you explicitly unlock it with a FREE or UNLOCK statement or until you close the file.

   - ALLOW NONE specifies no access to the record. This means that other access streams cannot retrieve the record unless they bypass lock checking with GET REGARDLESS.

   - ALLOW READ specifies read access to the record. This means that other access streams can retrieve the record, but cannot PUT or UPDATE the record.

   - ALLOW MODIFY specifies both read and write access to the record. This means that other access streams can GET, PUT, DELETE, or UPDATE the record.

2. When you do not specify an ALLOW clause, locking is imposed as follows:

   - If the file associated with *chnl-exp* was opened with UNLOCK EXPLICIT, BASIC imposes the ALLOW NONE lock on the retrieved record and the next GET or FIND does not unlock the previously locked record.
   - If the file associated with *chnl-exp* was not opened with UNLOCK EXPLICIT, BASIC locks the retrieved record and unlocks the previously locked record.

3. REGARDLESS specifies that the GET statement can override lock checking and read a record locked by another program.

4. REGARDLESS does not impose a lock on the retrieved record.

BASIC–PLUS–2 (BP2)

1. When you access a shared file, a successful GET locks the record or bucket and unlocks the previously locked record or bucket.

Examples

Sequential Access

```
100     MAP (XYZ) STRING LAST_NAME = 10, FIRST_NAME = 6
150     OPEN 'EMP.DAT' AS FILE #1, &
              ORGANIZATION SEQUENTIAL, MAP XYZ
200     GET #4
```

Random Access

```
100     MAP (XYZ) STRING LAST_NAME = 10, FIRST_NAME = 6
200     DECLARE RFA Address(99)
300     OPEN 'EMP.DAT' AS FILE #1,                          &
             ORGANIZATION SEQUENTIAL, MAP XYZ
400     GET #1
        Address(0) = GETRFA(1)
        !
        !
        !
500     GET #1, RFA Address(5)
        !
        !
        !
600     OPEN 'NEWEMP.DAT' AS FILE #2,                       &
             ORGANIZATION RELATIVE, MAP XYZ
700     GET #2, RECORD A%
        Address(A%) = GETRFA(2)
        !
        !
        !
        FIND #2, RFA Address(A%)
        !
        !
        !
```

(continued on next page)

```
900     OPEN 'OLDEMP.DAT' AS FILE #3,                    &
                ORGANIZATION INDEXED, MAP XYZ,           &
                PRIMARY KEY LAST_NAME
        GET #3, KEY #0 EQ "JONES"
        Address(5) = GETRFA(3)
        !
        !
        !
        GET #3, RFA Address(7)
```

VAX–11 BASIC

```
100     MAP (XYZ) STRING LAST_NAME = 10, FIRST_NAME = 6
300     OPEN 'EMP.DAT' AS FILE #1,                       &
                ORGANIZATION INDEXED,                    &
                MAP XYZ, PRIMARY KEY LAST_NAME,          &
                UNLOCK EXPLICIT
400     GET #1, KEY #0 EQ "JONES", ALLOW READ
```

25.0 GOSUB

Function

The GOSUB statement transfers control to a specified line number or label and stores the location of the GOSUB statement for eventual return from the subroutine.

Format

```
{ GO SUB }
{ GOSUB  }   target
```

Syntax Rules

1. *Target* must refer to an existing line number or label in the same program unit as the GOSUB statement or BASIC signals an error.
2. *Target* cannot be inside a FOR/NEXT, WHILE, or UNTIL loop or a multi-line function definition unless the GOSUB statement is also within that loop or function definition.

General Rules

None.

Examples

```
200     GOSUB 1100
        !
        !
        !
1100    ! Subroutine 1
        !
        !
        !
2100    RETURN
```

26.0 GOTO

Function

The GOTO statement transfers control to a specified line number or label.

Format

```
{ GO TO }
{ GOTO  }   target
```

Syntax Rules

1. *Target* must refer to an existing line number or label in the same program unit as the GOTO statement or BASIC signals an error.
2. *Target* cannot be inside a FOR/NEXT, WHILE, or UNTIL loop or a multi-line function definition unless the GOTO statement is also inside that loop or function definition.

General Rules

None.

Examples

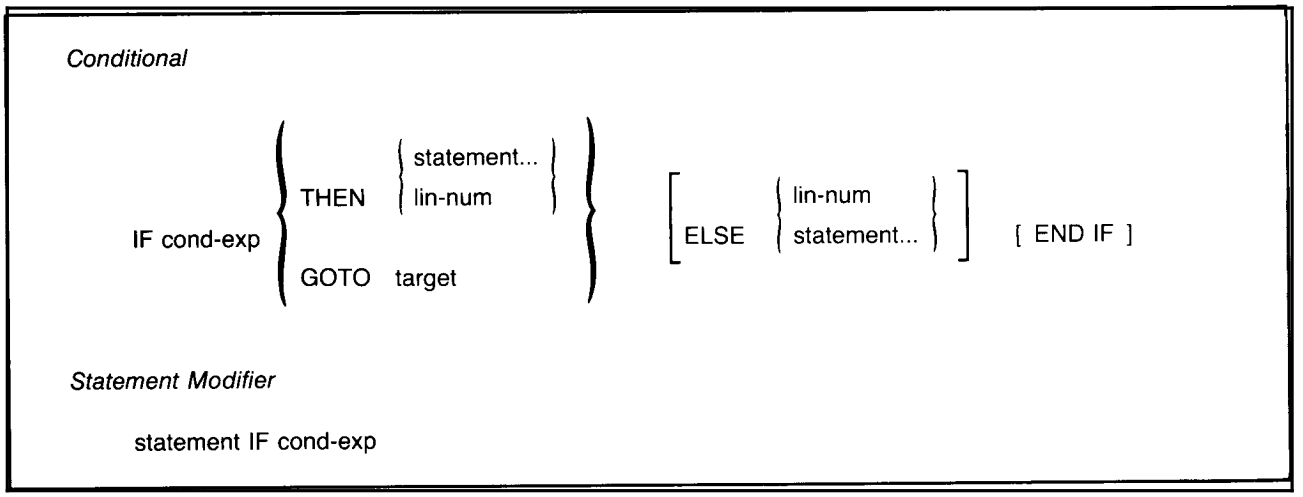
```
20      GOTO 200
```

27.0 IF

Function

The IF statement evaluates a conditional expression and transfers program control depending on the resulting value.

Format



Syntax Rules

Conditional

1. *Cond-exp* can be any valid conditional expression.
2. Any executable statement is valid in the THEN or ELSE clause, including another IF statement. You can include any number of statements in either clause.
3. All statements between the keyword THEN and the next ELSE, line number, or END IF are part of the THEN clause. All statements between the ELSE keyword and the next line number or END IF are part of the ELSE clause.
4. You can omit the THEN keyword when the target of a GOTO statement in the THEN is a *lin-num*. The THEN keyword is required when the target of a GOTO statement is a label.
5. BASIC assumes a GOTO statement when the ELSE keyword is followed by a *lin-num*. When the target of a GOTO statement is a label, the GOTO keyword is required.
6. If a THEN or ELSE clause contains a FOR, SELECT, UNTIL, or WHILE statement, then a corresponding NEXT or END statement must appear in the same THEN or ELSE clause.
7. IF statements can be nested to 12 levels.
8. The END IF statement terminates the most recent unterminated IF statement.
9. A new line number terminates all unterminated IF statements.

Statement Modifier

1. IF can modify any executable statement except a block statement such as FOR, WHILE, UNTIL, or SELECT.
2. *Cond-exp* can be any valid conditional expression.

General Rules

Conditional

1. BASIC evaluates *cond-exp* for truth or falsity. If true (nonzero), BASIC executes the THEN clause. If false (zero), BASIC skips the THEN clause and executes the ELSE clause, if present.
2. The NEXT keyword cannot be in a THEN or ELSE clause unless the IF statement associated with the NEXT keyword is also part of the THEN or ELSE clause.
3. Execution continues at the statement following the END IF or ELSE clause. If the statement does not contain an ELSE clause, execution continues at the next statement after the THEN clause.

Statement Modifier

1. BASIC executes *statement* only if the *cond-exp* is true (nonzero).

Examples

Conditional

```
19000    IF ERR = 11
         THEN
                 IF ERL = 1000
                 THEN GOTO ERROR_ROUTINE
                 ELSE
                         IF ERL = 2000
                         THEN 32700
                         ELSE
                                 IF ERL = 3000
                                 THEN GOTO ERROR_ROUTINE
                                 END IF
                         END IF
                 END IF
         ELSE PRINT ERT$(ERR)
         END IF
```

Statement Modifier

```
100      PRINT 'END OF PROCESSING' IF ERR = 11
```

28.0 INPUT

Function

The INPUT statement assigns values from your terminal or from a terminal-format file to program variables.

Format

```
INPUT [ chnl-exp, ] [ prompt ] vbl [ sep [ prompt ] vbl ]...

sep:      { , }
          { ; }

prompt:   str-const sep
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. *Vbl* cannot be a DEF function name unless the INPUT statement is inside the multi-line DEF that defines the function.

General Rules

1. The default *chnl-exp* is zero (the controlling terminal). If a *chnl-exp* is specified, a file must be open on that channel with ACCESS READ or MODIFY before the INPUT statement can execute.
2. You can include more than one *prompt* in an INPUT statement. The first *prompt* is issued for the first *vbl*, the second *prompt* for the second *vbl*, and so on. The *sep* that follows the *vbl* associated with the *prompt* has no formatting effect. BASIC always advances to a new line when you terminate input with a carriage return.
3. *Sep* in the *prompt* clause determines where the question mark is displayed and where the cursor is positioned for input.
   - A comma tells BASIC to skip to the next print zone and display the question mark. For example:

     ```
     100  INPUT 'NAME',YOUR_NAME$

     Run

     NAME      ?
     ```
   - A semicolon tells BASIC to display the question mark next to *str-const*. For example:

     ```
     100  INPUT 'ADDRESS';ADDR$

     Run

     ADDRESS?
     ```

4. BASIC signals an error if the INPUT statement has no argument.

5. If input comes from a terminal, BASIC displays the contents of *str-const*, if present, and a question mark (?). If you have not specified a *str-const*, BASIC displays only the question mark. The program then waits for data.

6. If the open channel does not correspond to a terminal, BASIC displays only the question mark.

7. When BASIC receives a line terminator or a complete record, it checks each data element for correct data type and range limits, then assigns the values to the corresponding variables.

8. If you specify a string variable to receive the input text, and the user enters an unquoted string in response to the prompt, BASIC ignores the string's leading and trailing spaces and tabs. An unquoted string cannot contain any commas.

9. When you enter several data elements in response to the INPUT prompt, you must separate them with commas.

10. If there is not enough data in the current record or line to satisfy the variable list, BASIC takes one of the following actions:

    - If the input device is a terminal, BASIC repeats the question mark, but not the *str-const*, on a new line until sufficient data is entered.
    - If the input device is not a terminal, BASIC signals "Not enough data in record" (ERR = 59).

11. If there are more data items than variables in the INPUT response, BASIC ignores the excess.

12. If there is an error in converting or assigning data (for example, assigning string data to a numeric variable), BASIC takes one of the following actions:

    - If the input device is a terminal, BASIC signals a warning, reexecutes the INPUT statement, and displays *str-const* and the question mark.
    - If the input device is not a terminal, BASIC signals "Illegal number" (ERR = 52) or "Data format error" (ERR = 50).

13. When a RESUME statement transfers control to an INPUT statement, the INPUT statement retrieves a new record regardless of any data left in the previous record.

14. After a successful INPUT statement, the RECOUNT variable contains the number of characters transferred from the file or terminal to the record buffer.

15. If you terminate input text with CTRL/Z, BASIC assigns the value to the variable and signals "End of file on device" (ERR=11) when the next terminal input statement executes. If there is no next INPUT, INPUT LINE, or LINPUT statement in the program, the CTRL/Z is passed to BASIC as a signal to exit the BASIC environment. BASIC signals "Unsaved changes have been made, CTRL/Z or EXIT to exit" if you have made changes to your program. If you have not made changes, BASIC exits from the BASIC environment and does not signal an error.

Examples

```
400     INPUT "TYPE IN 3 INTEGERS";A%, B%, C%

100     INPUT #3%, RECORD_STRING$

150     INPUT #1%, "PURCHASE NUMBER";PO_NUM%; "COST"; COST, "ID NUMBER";ID%
```

29.0 INPUT LINE

Function

The INPUT LINE statement assigns a string value, including the line terminator, from a terminal or terminal-format file to a string variable.

Format

```
INPUT LINE [ chnl-exp, ] [ prompt ] str-vbl [ sep [ prompt ] str-vbl ]...
```

sep: { , | ; }

prompt: str-const sep

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. *Vbl* cannot be a DEF function name unless the INPUT LINE statement is inside the multi-line DEF that defines the function.

General Rules

1. The default *chnl-exp* is zero (the controlling terminal). If a *chnl-exp* is specified, a file must be open on that channel with ACCESS READ before the INPUT LINE statement can execute.
2. You can include more than one *prompt* in an INPUT LINE statement. The first *prompt* is issued for the first *vbl*, the second *prompt* for the second *vbl*, and so on. The *sep* that follows the *vbl* associated with the *prompt* has no formatting effect. BASIC always advances to a new line when you terminate input with a carriage return.
3. *Sep* in the *prompt* clause determines where the question mark is displayed and where the cursor is positioned for input.
   - A comma tells BASIC to skip to the next print zone and display the question mark. For example:

```
100  INPUT LINE 'NAME',YOUR_NAME
Run
NAME      ?
```

   - A semicolon tells BASIC to display the question mark next to *str-const*. For example:

```
100  INPUT LINE 'ADDRESS';ADDR$
Run
ADDRESS?
```

4. BASIC signals an error if the INPUT LINE statement has no argument.
5. If input comes from a terminal, BASIC displays the contents of *str-const*, if present, and a question mark (?). If you have not specified a *str-const*, BASIC displays only the question mark. The program then waits for data.
6. If *chnl-exp* does not correspond to a terminal, BASIC displays only the question mark.
7. The INPUT LINE statement assigns all input characters including the line terminator(s) to *str-vbl*. Single and double quotation marks, commas, tabs, leading and trailing spaces, or other special characters in the string are part of the data.
8. When a RESUME statement transfers control to an INPUT LINE statement, the INPUT LINE statement retrieves a new record regardless of any data left in the previous record.
9. After a successful INPUT LINE statement, the RECOUNT variable contains the number of characters transferred from the file or terminal to the record buffer.
10. If you terminate input text with CTRL/Z, BASIC assigns the value to the variable and signals "End of file on device" (ERR=11) when the next terminal input statement executes. If there is no next INPUT, INPUT LINE, or LINPUT statement in the program, the CTRL/Z is passed to BASIC as a signal to exit the BASIC environment. BASIC signals "Unsaved changes have been made, CTRL/Z or EXIT to exit" if you have made changes to your program. If you have not made changes, BASIC exits from the BASIC environment and does not signal an error.

Examples

```
650     INPUT LINE "Type two words", Z$,"Type your name";N$

390     INPUT LINE #4%, RECORD_STRING$
```

30.0 ITERATE

Function

The ITERATE statement allows you to explicitly reexecute a loop.

Format

```
ITERATE [ label ]
```

Syntax Rules

1. *Label* is the label of the first statement of a FOR–NEXT, WHILE, or UNTIL loop
2. The ITERATE statement can be used only within a FOR–NEXT, WHILE, or UNTIL loop.

General Rules

1. ITERATE is equivalent to an unconditional branch to the current loop's NEXT statement. If you supply a *label*, ITERATE transfers control to the NEXT statement in the specified loop. If you do not supply a *label*, ITERATE transfers control to the current loop's NEXT statement.
2. *Label* must conform to the rules for naming variables.

Examples

```
1000    Date_loop:  WHILE 1% = 1%
                        GET #1
                        ITERATE Date_loop IF Day$ <> Today$
                        ITERATE Date_loop IF Month$ <> This_month$
                        ITERATE Date_loop IF Year$ <> This_year$
                        PRINT Item$
                    NEXT
```

31.0 KILL

Function

The KILL statement deletes a disk file, removes the file's directory entry, and releases the file's storage space.

Format

```
KILL file-spec
```

Syntax Rules

1. *File-spec* can be a quoted string constant, a string variable, or a string expression. It cannot be an unquoted string constant.

General Rules

1. The KILL statement marks a file for deletion but does not delete the file until all users have closed it.
2. If you do not specify a complete *file-spec*, BASIC uses the default device and directory. If you do not specify a file version, *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* delete the highest version of the file.
3. The *file-spec* must exist or BASIC signals an error.
4. You can delete a file in another directory if you have access to that directory and privilege to delete the file.

Examples

```
200     KILL "TEMP.DAT"
```

32.0 LET

Function

The LET statement assigns a value to one or more variables.

Format

```
[LET] vbl,... = exp
```

Syntax Rules

1. *Vbl* cannot be a DEF or FUNCTION name unless the LET statement occurs inside that DEF block or in that FUNCTION subprogram.
2. You cannot assign string data to a numeric variable or numeric data to a string variable.
3. The keyword LET is optional.

General Rules

1. When you assign a value to a subscripted variable, BASIC evaluates the subscripts from left to right before evaluating *exp* and assigning the value. In the following example, line 10 assigns the value 5 to I, then line 20 assigns the value 2 to A(5) and to I:

```
10      LET I = 5
20      LET A(I),I = 2
```

2. The value assigned to a numeric variable is converted to the variable's data type. For example, if you assign a floating-point value to an integer variable, BASIC truncates the value to an integer.
3. For dynamic strings, the destination string's length equals the source string's length.
4. When you assign a value to a fixed-length string variable, the value is left-justified and padded with spaces or truncated to match the length of the string variable.

Examples

```
10      LET A = 3.141
20      A$ = "ABCDEFG"
```

33.0 LINPUT

Function

The LINPUT statement assigns a string value, without line terminators, from a terminal or terminal-format file to a string variable.

Format

```
LINPUT [ chnl-exp, ] [ prompt ] str-vbl [ sep [ prompt ] str-vbl ]...

sep:      { , }
          { ; }

prompt:   str-const sep
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

2. *Vbl* cannot be a DEF function name unless the LINPUT statement is inside the multi-line DEF that defines the function.

General Rules

1. The default *chnl-exp* is zero (the controlling terminal). If you specify a *chnl-exp*, the file associated with that channel must have been opened with ACCESS READ or MODIFY.

2. You can include more than one *prompt* in an INPUT LINE statement. The first *prompt* is issued for the first *vbl*, the second *prompt* for the second *vbl*, and so on. The *sep* that follows the *vbl* associated with the *prompt* has no formatting effect. BASIC always advances to a new line when you terminate input with a carriage return.

3. *Sep* in the *prompt* clause determines where the question mark is displayed and where the cursor is positioned for input.

   - A comma tells BASIC to skip to the next print zone and display the question mark. For example:

```
100     LINPUT "NAME",YOUR_NAME
Run
NAME    ?
```

(continued on next page)

- A semicolon tells BASIC to display the question mark next to *str-const*. For example:

```
100 LINPUT "ADDRESS";ADDR$
Run
ADDRESS?
```

4. BASIC signals an error if the LINPUT statement has no argument.
5. If input comes from a terminal, BASIC displays the contents of *str-const*, if present, and a question mark (?). If you have not specified a *str-const*, BASIC displays only the question mark. The program then waits for data.
6. If *chnl-exp* does not correspond to a terminal, BASIC displays only the question mark.
7. The LINPUT assigns all characters except the line terminator(s) to *str-vbl*. Single and double quotation marks, commas, tabs, leading and trailing spaces, or other special characters in the string are part of the data.
8. If the RESUME statement transfers control to a LINPUT statement, the LINPUT statement retrieves a new record regardless of any data left in the previous record.
9. After a successful LINPUT statement, the RECOUNT variable contains the number of bytes transferred from the file or terminal to the record buffer.
10. If you terminate input text with CTRL/Z, BASIC assigns the value to the variable and signals "End of file on device" (ERR = 11) when the next terminal input statement executes. If there is no next INPUT, INPUT LINE, or LINPUT statement in the program, the CTRL/Z is passed to BASIC as a signal to exit the BASIC environment.

Examples

```
100     LINPUT "ENTER YOUR LAST NAME";Last_name$
200     LINPUT #2%, Last_name$
```

34.0 LSET

Function

The LSET statement assigns left-justified data to a string variable. LSET does not change the length of the destination string variable.

Format

```
LSET str-vbl,... = str-exp
```

Syntax Rules

1. *Str-vbl* is the destination string. *Str-exp* is the string value assigned to *str-vbl*.
2. BASIC evaluates *str-vbl's* subscripts (if present) before evaluating *str-exp*.
3. *Str-vbl* cannot be a DEF function name unless the LSET statement is inside the multi-line DEF that defines the function.

General Rules

1. The LSET statement treats all strings as fixed-length. LSET neither changes the length of the destination string nor creates new storage. Rather, it overwrites the *str-vbl's* current storage.
2. If the destination string is longer than *str-exp*, LSET left-justifies *str-exp* and pads it with spaces on the right. If smaller, LSET truncates characters from the right of *str-exp* to match the length of *str-vbl*.

Examples

```
10      LSET ALPHA$='XYZ'

20      LSET A$, B$ = CODE$ + NA_ME$
```

35.0 MAP

Function

The MAP statement defines a named area of statically allocated storage called a PSECT, declares data fields in the record, and associates them with program variables.

Format

```
MAP (map-nam) { [ data-type ] map-item },...

map-item:  { num-unsubs-vbl-nam
           { num-array-nam ( int-const,... )
           { str-unsubs-vbl-nam [ = int-const ]
           { str-array-nam ( int-const,... ) [ = int-const ]
           { FILL [ ( int-const ) ] [ = int-const ]
           { FILL% [ ( int-const ) ]
           { FILL$ [ ( int-const ) ] [ = int-const ]
```

Syntax Rules

1. *Map-nam* is global to the program and task. It cannot appear elsewhere in the program unit as a variable name.

VAX

2. In *VAX–11 BASIC*, *map-nam* can consist of from 1 to 31 characters. The first character of the name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), periods (.), or underscores (_).

BP2

3. In *BASIC–PLUS–2*, *map-nam* can consist of from one to six characters. The first character must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).

4. *Map-item* declares the name and format of the data to be stored.

   - *Num-unsubs-vbl-nam* and *num-arr-nam* specify a numeric variable or a numeric array.
   - *Str-unsubs-vbl-nam* and *str-arr-nam* specify a fixed-length string variable or array. You can specify the number of bytes to be reserved for the variable with the *=int-const* clause. The default string length is 16.
   - The FILL, FILL%, and FILL$ keywords allow you to reserve parts of the record buffer within or between data elements and to define the format of the storage. *Int-const* specifies the number of FILL items to be reserved. The *=int-const* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 21 describes FILL item format and storage allocation.

Note

In the applicable formats of FILL, *(int-const)* represents a repeat count, not an array subscript. FILL (n), for example, represents n elements, not n + 1.

5. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined by a RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2.
6. When you specify a *data-type*, all following *map-items*, including FILL items, are of that data type until you specify a new data type.
7. If you do not specify any *data-type, map-items* without a data-typing suffix character (% or $) take the current default data type and size.
8. Variable names, array names, and FILL items following a *data-type* cannot end in a dollar sign or percent sign.
9. Variables and arrays declared in a MAP statement cannot be declared elsewhere in the program by any other declarative statements.

General Rules

1. BASIC does not execute MAP statements. The MAP statement allocates static storage and defines data at compile time.
2. A program can have multiple MAPs with the same name. The allocation for each MAP overlays the others. Thus, data is accessible in many ways. The actual size of the data area is the size of the largest MAP. When you link or task-build your program, the size of the MAP area is the size of the largest MAP with that name.
3. *Map-items* with the same name can appear in different MAP statements with the same *map-nam* only if they match exactly in attributes such as data type, position, and so forth. If the attributes are not the same, BASIC signals an error. For example:

```
100     MAP (ABC) LONG A, B
200     MAP (ABC) LONG A, C ! This MAP statement is valid
300     MAP (ABC) LONG B, A ! This MAP statement produces an error
400     MAP (ABC) WORD A, B ! This MAP statement produces an error
```

   Line 300 causes BASIC to signal the error "variable <name> not aligned in multiple references in MAP <name>", while line 400 generates the error "attributes of overlaid variable <name< don't match".
4. The MAP statement should precede any reference to variables declared in it.
5. Storage space for *map-items* is allocated in order of occurrence in the MAP statement.
6. A MAP area can be accessed by more than one program module, as long as you define the *map-nam* in each module that references the MAP.
7. A COMMON area and a MAP area with the same name specify the same storage area and are not allowed in the same program module.

8. A MAP named in an OPEN statement's MAP clause is associated with that file. The file's records and record fields are defined by that MAP. The size of the MAP determines the record size for file I/O, unless the OPEN statement includes a RECORDSIZE clause.
9. *VAX–11 BASIC* does not initialize variables in the MAP statement.
10. *BASIC–PLUS–2* initializes MAP variables to zero or a null string.

Examples

```
200     MAP (BUF1) BYTE AGE, STRING EMP_NAME = 20, SINGLE EMP_NUM

400     MAP (BUF1) BYTE FILL, STRING LAST_NAME = 12, FILL = 8, SINGLE FILL
```

36.0 MAP DYNAMIC

Function

The MAP DYNAMIC statement names the variables and arrays whose size and position in a MAP buffer can change at run time. BASIC sets all variable and array element pointers to the beginning of the MAP buffer when the MAP DYNAMIC statement is processed.

Format

```
MAP DYNAMIC (map-nam) { [ data-type ] map-item },...

map-item:   { num-unsubs-vbl-nam
            { num-array-nam ( int-const,... )
            { str-unsubs-vbl-nam
            { str-array-nam ( int-const,... )
```

Syntax Rules

1. *Map-nam* is the storage area named in a MAP statement.

2. *Map-item* declares the name and data type of the items to be stored in the map buffer. All variable pointers point to the beginning of the map buffer until the program executes a REMAP statement.

   - *Num-unsubs-vbl-nam* and *num-arr-nam* specify a numeric variable or a numeric array.
   - *Str-unsubs-vbl-nam* and *str-arr-nam* specify a string variable or array. You cannot specify the number of bytes to be reserved for the variable in the MAP DYNAMIC statement. All string items have a fixed-length of zero until the program executes a REMAP statement.

3. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in the RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.

4. When you specify a *data-type*, all following *map-items* are of that data type until you specify a new data type.

5. If you do not specify any *data-type, map-items* take the current default data type and size.

6. Variable names and array names following a *data-type* cannot end in a dollar sign or percent sign suffix character.

7. Variables and arrays declared in a MAP DYNAMIC statement cannot be declared elsewhere in the program by any other declarative statements.

8. *Map-items* must be separated with commas.

MAP DYNAMIC

General Rules

1. The MAP DYNAMIC statement does not affect the amount of storage allocated to the map buffer declared in a previous MAP statement. Until your program executes a REMAP statement, all variable and array element pointers point to the beginning of the MAP buffer.
2. BASIC does not execute MAP DYNAMIC statements. The MAP DYNAMIC statement names the variables whose size and position in the MAP buffer can change and defines their data type.
3. If there is no MAP statement in the program unit with the same *map-nam* specified in the MAP DYNAMIC statement, BASIC signals the error "Insufficient space for MAP DYNAMIC variables in MAP <name>".
4. The MAP DYNAMIC statement must lexically precede the REMAP statement or BASIC signals the error "MAP variable <name> referenced before declaration".

Examples

```
100     MAP (MY.BUF) STRING DUMMY = 512
        MAP DYNAMIC (MY.BUF) STRING LAST, FIRST, MIDDLE,        &
                             BYTE AGE, STRING EMPLOYER,         &
                             STRING CHARACTERISTICS
```

37.0 MARGIN (VAX–11 BASIC)

Function

The MARGIN statement specifies the margin width for a terminal or for records in a terminal-format file.

Format

```
MARGIN [ chnl-exp, ] int-exp
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. *Int-exp* specifies the margin width.

General Rules

1. If you do not specify a *chnl-exp*, BASIC sets the margin on the controlling terminal.
2. The file associated with *chnl-exp* must be an open terminal-format file.
3. BASIC signals the error "Illegal operation" (ERR = 141) if the file associated with *chnl-exp* is not a terminal-format file.
4. If *chnl-exp* does not correspond to a terminal, and if *int-exp* is zero, BASIC sets the right margin to the size specified by the RECORDSIZE clause in the OPEN statement, if present. If no RECORDSIZE clause is present, BASIC sets the margin to the RMS blocksize.
5. If *chnl-exp* is not present or if it corresponds to a terminal, and if *int-exp* is zero, BASIC sets the right margin to the size specified by the RECORDSIZE clause in the OPEN statement, if present. If no RECORDSIZE clause is present, BASIC sets the margin to the default terminal width.
6. BASIC prints as much of a specified record as the margin setting allows on one line before going to a new line. Numeric fields are never split across lines.
7. If you specify a margin larger than the channel's recordsize, BASIC signals an error.
8. The MARGIN statement is in effect only while *chnl-exp* is open. When you close *chnl-exp*, BASIC returns to the default margin when you reopen the channel.

Examples

```
30      MARGIN #4, 132%
```

38.0 MAT

Function

The MAT statement lets you implicitly create and manipulate one- and two-dimensional arrays. You can use the MAT statement to assign values to array elements or to redimension a previously dimensioned array. You can also perform matrix arithmetic operations such as multiplication, addition, and subtraction, and other matrix operations such as transposing and inverting matrices.

Format

Initialization (Numeric)

```
MAT num-array = { CON } [ ( int-exp1 [, int-exp2 ] ) ]
                { IDN }
                { ZER }
```

Initialization (String)

```
MAT str-array = NUL$ [ ( int-exp1 [, int-exp2 ] ) ]
```

Array Arithmetic

```
MAT num-array1 = num-array2 [ { + } num-array3 ]
                            { - }
                            { * }
```

Scalar Multiplication

```
MAT num-array4 = ( num-exp ) * num-array5
```

Inversion and Transposition

```
MAT num-array6 = { TRN } ( num-array7 )
                 { INV }
```

Syntax Rules

1. You cannot use the MAT statement on arrays of more than two dimensions.
2. In *VAX–11 BASIC*, you cannot use the MAT statement on arrays of data-type DECIMAL or on arrays named in a RECORD statement.
3. When initializing arrays, you can specify the array bounds. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the new dimensions of an existing array.

4. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
   - If you do not specify bounds, BASIC creates the array and dimensions it to (10) or (10,10).
   - If you do specify bounds, BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC signals "Redimensioned array" (ERR=105).
5. To perform MAT operations on arrays larger than (10,10), create the input and output arrays with the DIM statement.
6. When the array exists, the following rules apply:
   - If you specify bounds, BASIC redimensions the array to the specified size. However, MAT operations cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC does not redimension the array.
7. An array passed to a subprogram and redimensioned there by a MAT statement remains redimensioned when control returns to the calling program, with two exceptions:
   - When the array is within a RECORD and is passed BY DESC.
   - When the array is passed BY REF.

Initialization

1. CON sets all elements of *num-array* to one, except those in row and column zero.
2. IDN creates an identity matrix from *num-array*. The number of rows and columns in *num-array* must be identical. IDN sets all elements to zero except those on the diagonal from *num-array*(1,1) to *num-array*(n,n), which are set to one.
3. ZER sets all array elements to zero, except those in row and column zero.
4. NUL$ sets all elements of a string array to the null string, except those in row and column zero.

Array Arithmetic

1. The equals sign (=) assigns the results of the specified operation to the elements in *num-array1*.
2. If *num-array3* is not specified, BASIC assigns the values of *num-array2*'s elements to the corresponding elements of *num-array1*. *Num-array1* must have at least as many rows and columns as *num-array2*.
3. Use the plus sign (+) to add the elements of two arrays. *Num-array2* and *num-array3* must have identical bounds.
4. Use the minus sign (–) to subtract the elements of two arrays. *Num-array2* and *num-array3* must have identical bounds.
5. Use the asterisk (*) to perform matrix multiplication on the elements of *num-array2* and *num-array3* and to assign the results to *num-array1*. This operation gives the dot product of *num-array2* and *num-array3*. All three arrays must be two-dimensional, and the number of columns in *num-array2* must equal the number of rows in *num-array3*. BASIC redimensions *num-array1* to have the same number of rows as *num-array2* and the same number of columns as *num-array3*.

Scalar Multiplication

1. BASIC multiplies each element of *num-array5* by *num-exp* and stores the results in the corresponding elements of *num-array4*.

Inversion and Transposition

1. TRN transposes *num-array7* and assigns the results to *num-array6*. If *num-array7* has m rows and n columns, *num-array6* will have n rows and m columns. Both arrays must be two-dimensional.
2. You cannot transpose a matrix to itself: MAT A = TRN(A) is invalid.
3. INV inverts *num-array7* and assigns the results to *num-array6*. *Num-array7* must be a two-dimensional array that can be reduced to the identity matrix using elementary row operations. The row and column dimensions must be identical.

General Rules

1. You cannot increase the number of array elements or change the number of dimensions in an array when you redimension with the MAT statement. That is, you can redimension an array with dimensions (5,4) to (4,5) or (3,2), but you cannot redimension that array to (5,5) or to (10). The total number of array elements includes those in row and column zero.
2. If an array is named in both a DIM statement and a MAT statement, the DIM statement must lexically precede the MAT statement.
3. MAT statements do not operate on elements in: 1) the zero element (one-dimensional arrays) or 2) the zero row or column (two-dimensional arrays). MAT statements use these elements to store results of intermediate calculations. Therefore, you should not depend on values in row and column zero if your program uses MAT statements.

Examples

Initialization (Numeric)

```
100      MAT CONVERT = ZER(10,10)
```

Initialization (String)

```
1000     MAT NA_ME$ = NUL$(5,5)
```

Array Arithmetic

```
2000     MAT NEW_INT = OLD_INT - RSLT_INT
```

Scalar Multiplication

```
3000     MAT Z40 = (4.24) * Z
```

Inversion and Transposition

```
4000     MAT Q% = INV (Z%)
```

39.0 MAT INPUT

Function

The MAT INPUT statement assigns values from a terminal or terminal-format file to array elements.

Format

```
MAT INPUT [ chnl-exp, ] { array [ ( int-exp1 [, int-exp2 ] ) ] },...
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. The file associated with *chnl-exp* must be an open terminal-format file. If *chnl-exp* is not specified, BASIC takes data from the controlling terminal.
3. You cannot use the MAT INPUT statement on arrays of more than two dimensions.
4. In *VAX–11 BASIC*, you cannot use the MAT INPUT statement on arrays of data-type DECIMAL or on arrays named in a RECORD statement.
5. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
6. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
   - If you do not specify bounds, BASIC creates the array, dimensions it to (10,10), and prompts only for the first array element.
   - If you do specify bounds, BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC signals "Redimensioned array" (ERR=105).
7. To MAT INPUT to arrays larger than (10,10), create the input and output arrays with the DIM statement. When the array exists, the following rules apply:
   - If you specify bounds, BASIC redimensions the array to the specified size. However, MAT INPUT cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC does not redimension the array.

General Rules

1. The MAT INPUT statement prompts with a question mark on terminals open on channel zero only.
2. Use commas to separate data elements and a line terminator to end the input of data. Use an ampersand before the line terminator to input data on more than one line.
3. The MAT INPUT statement assigns values by row. That is, it assigns values to all elements in row one before beginning row two.

4. The MAT INPUT statement assigns the row number of the last data element transferred into the array to the system variable, NUM.
5. The MAT INPUT statement assigns the column number of the last data element transferred into the array to the system variable, NUM2.
6. If there are fewer elements in the input data than there are array elements, BASIC does not change the remaining array elements.
7. If there are more data elements in the input stream than there are array elements, BASIC ignores the excess.
8. Row zero and column zero are not changed.

Examples

```
1000  MAT INPUT EMP_NAME$(10,10)
```

40.0 MAT LINPUT

Function

The MAT LINPUT statement receives string data from a terminal or terminal-format file and assigns it to string array elements.

Format

```
MAT LINPUT [ chnl-exp, ] { str-array [ ( int-exp1 [, int-exp2 ] ) ] },...
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. You cannot use the MAT LINPUT statement on arrays of more than two dimensions.
3. In *VAX–11 BASIC*, you cannot use the MAT LINPUT statement on arrays of data-type DECIMAL or on arrays named in a RECORD statement.
4. The file associated with *chnl-exp* must be an open terminal-format file. If *chnl-exp* is not specified, BASIC takes data from the controlling terminal.
5. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
6. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
   - If you do not specify bounds, BASIC creates the array, dimensions it to (10,10), and prompts only for the first array element.
   - If you do specify bounds, BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC signals "Redimensioned array" (ERR=105).
7. To MAT LINPUT to arrays larger than (10,10), create the input and output arrays with the DIM statement.
8. When the array exists, the following rules apply:
   - If you specify bounds, BASIC redimensions the array to the specified size. However, MAT LINPUT cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC does not redimension the array.

General Rules

1. For terminals open on channel zero only, the MAT LINPUT statement prompts with a question mark for each string array element, starting with element (1,1). BASIC assigns values to all elements of row one before beginning row two.
2. The MAT LINPUT statement assigns the row number of the last data element transferred into the array to the system variable, NUM.

3. The MAT LINPUT statement assigns the column number of the last data element transferred into the array to the system variable, NUM2.
4. Typing only a line terminator in response to the question mark prompt causes BASIC to assign a null string to that string array element.
5. MAT LINPUT does not change row and column zero.

Examples

```
400   MAT LINPUT TIME_CARD$(10%)
```

41.0 MAT PRINT

Function

The MAT PRINT statement prints the contents of a one- or two-dimensional array on your terminal or assigns the value of each array element to a record in a terminal-format file.

Format

```
MAT PRINT [ chnl-exp, ] { array [ ( int-exp1 [, int-exp2 ] ) ] [ sep ] }...

sep:  { , }
      { ; }
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

2. You cannot use the MAT PRINT statement on arrays of more than two dimensions.

3. In *VAX–11 BASIC*, you cannot use the MAT PRINT statement on arrays of data-type DECIMAL or on arrays named in a RECORD statement.

4. The file associated with *chnl-exp* must be an open terminal-format file. If you do not specify a *chnl-exp*, BASIC takes data from the controlling terminal.

5. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.

6. If the array does not exist, the following rules apply:

   - If you do not specify bounds, BASIC creates the array and dimensions it to (10,10).
   - If you do specify bounds, BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC prints (10) or (10,10) elements and signals "Subscript out of range" (ERR = 55).

7. When the array exists, the following rules apply:

   - If the specified bounds are smaller than the maximum bounds of a dimensioned array, BASIC prints a subset of the array, but does not redimension the array. For example, if you use the DIM statement to dimension A(20,20), and then MAT PRINT A(2,2), BASIC prints elements (1,1), (1,2), (2,1), and (2,2) only; array A(20,20) does not change.
   - If you do not specify bounds, BASIC prints the entire array.

8. *Sep* determines the output format for the array:
   - If you use a comma, BASIC prints each array element in a new print zone and starts each row on a new line.
   - If you use a semicolon, BASIC separates each array element with a space and starts each row on a new line.
   - If you do not use a *sep* character, BASIC prints each array element on its own line.
9. When you use the MAT PRINT statement to print more than one array, each array name except the last must be followed with either a comma or a semicolon. BASIC prints a blank line between arrays.

General Rules

1. The MAT PRINT statement does not print elements in row or column zero.
2. The MAT PRINT statement cannot redimension an array.

Examples

```
500   MAT PRINT #1, TIME_CARD$(25);
```

42.0 MAT READ

Function

The MAT READ statement assigns values from DATA statements to array elements.

Format

```
MAT READ { array [ ( int-exp1 [, int-exp2 ] ) ] },...
```

Syntax Rules

1. You cannot use the MAT READ statement on arrays of more than two dimensions.
2. In *VAX–11 BASIC*, you cannot use the MAT READ statement on arrays of data-type DECIMAL or on arrays named in a RECORD statement.
3. *Int-exp1* and *int-exp2* define the upper bounds of the array being implicitly created or the dimensions of an existing array.
4. If you are creating an array, *int-exp1* and *int-exp2* cannot exceed 10.
   - If you do not specify bounds, BASIC creates the array and dimensions it to (10,10).
   - If you do specify bounds, BASIC creates the array with the specified bounds. If the bounds exceed (10) or (10,10), BASIC signals "Redimensioned array" (ERR=105).
5. To MAT READ arrays larger than (10,10), create the array with the DIM statement.
6. When the array exists, the following rules apply:
   - If you specify bounds, BASIC redimensions the array to the specified size. However, MAT READ cannot increase the total number of array elements.
   - If you do not specify bounds, BASIC does not redimension the array.

General Rules

1. The DATA statement(s) must be in the same program unit as the MAT READ statement.
2. The MAT READ statement assigns data items by row. That is, it assigns data items to all elements in row one before beginning row two.
3. The MAT READ statement does not read elements into row or column zero.
4. The MAT READ statement assigns the row number of the last data element transferred into the array to the system variable, NUM.

5. The MAT READ statement assigns the column number of the last data element transferred into the array to the system variable, NUM2.

6. If you MAT READ an existing array without specifying bounds, BASIC does not redimension the array. If you MAT READ an existing array and specify bounds, BASIC redimensions the array.

Examples

```
100   MAT READ Z%
```

43.0 MOVE

Function

The MOVE statement transfers data between a record buffer and a list of variables.

Format

```
MOVE { TO   } chnl-exp, move-item,...
     { FROM }

move-item: { num-vbl
           { num-array ( [,]... )
           { str-vbl [ = int-exp ]
           { str-array ( [,]... ) [ = int-exp ]
           { [ data-type ] FILL [ ( int-exp ) ] [ = int-const ]
           { FILL% [ ( int-exp ) ]
           { FILL$ [ ( int-exp ) ] [ = int-exp ]
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. *Move-item* specifies the variable or array to which or from which data is to be moved.
3. *Num-vbl* and *num-array* specify a numeric variable or a numeric array. Parentheses indicate the number of dimensions in a numeric array. The number of dimensions is equal to the number of commas plus one. That is, empty parentheses indicate a one-dimensional array, one comma indicates a two-dimensional array, and so on.
4. *Str-vbl* and *str-array* specify a fixed length string variable or array. Parentheses indicate the number of dimensions in a string array. The number of dimensions is equal to the number of commas plus one. You can specify the number of bytes to be reserved for the variable or array elements with the *=int-exp* clause. The default string length for a MOVE FROM statement is 16. For a MOVE TO statement, the default is the string's length.
5. The FILL, FILL%, and FILL$ keywords allow you to transfer fill items of a specific data type. Table 21 shows FILL item formats, representations, and storage requirements.
   - If you specify a *data-type* before the FILL keyword, the fill is of that data type. If you do not specify a *data-type*, the fill is of the default data type. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined by a RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual. FILL items following a *data-type* cannot end in a dollar sign or percent sign.
   - *Int-exp* specifies the number of FILL items to be moved.
   - FILL% indicates integer fill. FILL$ indicates string fill. The *=int-exp* clause specifies the number of bytes to be moved for string FILL items.

Note

In the applicable formats of FILL, *(int-exp)* represents a repeat count, not an array subscript. FILL (n), for example, represents n elements, not n + 1.

6. You cannot use an expression or function reference as a *move-item*.

General Rules

1. Before a MOVE FROM statement can execute, the file associated with *chnl-exp* must be open and there must be a record in the record buffer.
2. A MOVE statement neither transfers data to or from external devices, nor invokes the system Record Management Services. Instead, it transfers data between user areas. Thus, a record should first be fetched with the GET statement before using a MOVE FROM, and a MOVE TO should be followed by a PUT or UPDATE statement that writes the record to a file.
3. MOVE FROM transfers data from the record buffer to the *move-item*.
4. MOVE TO transfers data from the *move-item* to the record buffer.
5. The MOVE statement does not affect the record buffer's size. If a MOVE statement partially fills a buffer, the rest of the buffer is unchanged. If there is more data in the variable list than in the buffer, BASIC signals "MOVE overflows buffer" (ERR=161).
6. Each MOVE statement to or from a channel transfers data starting at the beginning of the buffer. For example:

```
200      MOVE FROM #1%, I%, A$ = I%
```

   In this example, BASIC assigns the first value in the record buffer to I%; the value of I% is then used to determine the length of A$.

7. If a MOVE statement operates on an entire array:
   - BASIC transfers elements of row and column zero (contrast this with the MAT statements).
   - The storage size of the array elements and the size of the array determine the amount of data moved. A MOVE statement that transfers data from the buffer to a longword integer array transfers the first four bytes of data into element (0,0), the next four bytes of data into element (0,1), and so on.
8. If the MOVE TO statement specifies an explicit string length, the following restrictions apply:
   - If the string is equal to or longer than the explicit string length, BASIC moves only the specified number of characters into the buffer.
   - If the string is shorter than the explicit string length, BASIC moves the entire string and pads it with spaces to the specified length.
9. BASIC does not check the validity of data during the MOVE operation.

Examples

```
990    MOVE FROM #4%, RUNS%, HITS%, ERRORS%, RBI%, BAT_AVERAGE
100    MOVE TO #9%, FILL$ = 10%, A$ = 10%, B$ = 30%, C$ = 2%
```

44.0 NAME AS

Function

The NAME AS statement changes the name of a specified file.

Format

```
NAME file-spec1 AS file-spec2
```

Syntax Rules

1. *File-spec1* and *file-spec2* must be string expressions.
2. There is no default for file type in *file-spec1* or *file-spec2*. If the file to be renamed has a file type, *file-spec1* must include both the file name and the file type. If you specify only a file name, BASIC searches for a file with no file type. If you do not specify a file type for *file-spec2*, BASIC names the file, but does not assign a file type.

General Rules

1. If the file specified by *file-spec1* does not exist, BASIC signals "Can't find file or account" (ERR = 5).
2. In *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems*, file version numbers are optional. BASIC renames the highest version of *file-spec1* if you do not specify a version number.
3. In *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems*, if you use the NAME AS statement on an open file, BASIC does not rename the file until you close it.

Examples

```
400   NAME "OUT.DAT" AS "RERUN.DAT"

500   NAME OLD_FILE$ AS NEW_FILE$
```

45.0 NEXT

Function

The NEXT statement marks the end of a FOR, UNTIL, or WHILE loop.

Format

```
NEXT [ num-unsubs-vbl ]
```

Syntax Rules

1. *Num-unsubs-vbl* is required in a FOR loop and must correspond to the *num-unsubs-vbl* specified in the FOR statement.
2. *Num-unsubs-vbl* is not allowed in an UNTIL or WHILE loop.

General Rules

1. Each NEXT statement must have a corresponding FOR, UNTIL, or WHILE statement or BASIC signals an error.

Examples

```
100   NEXT I%
```

46.0 NOMARGIN (VAX–11 BASIC)

Function

The NOMARGIN statement removes the right margin limit set with the MARGIN statement for a terminal or a terminal-format file.

Format

```
NOMARGIN [ chnl-exp ]
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

General Rules

1. When you specify NOMARGIN, the right margin is set to 132.
2. *Chnl-exp*, if specified, must be an open terminal-format file or a terminal.
3. If you do not specify a *chnl-exp*, BASIC sets the margin on the controlling terminal to 132.
4. The NOMARGIN statement applies to the specified channel only while the channel is open. If you close the channel and then reopen it, BASIC uses the default margin of 72.

Examples

```
1000  NOMARGIN #2%
```

47.0 ON ERROR GO BACK

Function

After BASIC executes an ON ERROR GO BACK in a subprogram or DEF, control transfers to the calling program when an error occurs.

Format

```
{ ONERROR  }
{ ON ERROR }   GO BACK
```

Syntax Rules

None.

General Rules

1. The ON ERROR GO BACK statement is the default error handler for DEF functions.
2. An ON ERROR GO BACK statement executed in the main program is equivalent to an ON ERROR GOTO 0 statement.
3. If a main program calls a subprogram named SUB1, and SUB1 calls the subprogram named SUB2, an ON ERROR GO BACK statement executed in SUB2 transfers control to SUB1 when an error occurs in SUB2. If SUB1 also has executed an ON ERROR GO BACK statement, BASIC transfers control to the main program's error handling routine.
4. If there is no error outstanding, execution of an ON ERROR GO BACK statement causes subsequent errors to return control to the calling program's error handler.
5. If there is an error outstanding, execution of an ON ERROR GO BACK statement immediately transfers control to the calling program's error handler.
6. The ON ERROR GO BACK statement remains in effect until the program unit completes execution or until BASIC executes another ON ERROR statement.

Examples

```
100       SUB LIST (A$)
          ON ERROR GOTO 19000
          OPEN A$ FOR INPUT AS FILE #1
400       LINPUT #1, B$
          PRINT B$
600       GOTO 400
          !
          !
          !
19000     IF (ERR = 11%) AND (ERL = 400%)
                    THEN CLOSE #1%
                    RESUME 32767
                    ELSE ON ERROR GO BACK
32767     SUBEND
```

48.0 ON ERROR GOTO

Function

The ON ERROR GOTO statement transfers program control to a specified line or label in the current program unit when an error occurs.

Format

```
{ ONERROR  }   { GO TO }
{ ON ERROR }   { GOTO  }   target
```

Syntax Rules

1. *Target* must exist in the same program unit as the ON ERROR GOTO statement.
2. If an ON ERROR GOTO is in a DEF, *target* must also be in that function definition.

General Rules

1. Execution of an ON ERROR GOTO statement causes subsequent errors to transfer control to the specified *target*.
2. The ON ERROR GOTO statement remains in effect until the program unit completes execution or until BASIC executes another ON ERROR statement.
3. BASIC does not allow recursive error handling. If a second error occurs during execution of an error-handling routine, control passes to the BASIC error handler and the program stops executing.

Examples

```
500     ON ERROR GOTO 9999

600     ON ERROR GOTO YES_ROUTINE
```

49.0 ON ERROR GOTO 0

Function

The ON ERROR GOTO 0 statement disables user error handling and passes control to the BASIC error handler when an error occurs.

Format

```
                 { GO TO }
ON ERROR         { GOTO  }    0
```

Syntax Rules

None.

General Rules

1. If an error is outstanding, execution of an ON ERROR GOTO 0 statement immediately transfers control to the BASIC error handler.
2. If there is no error outstanding, execution of an ON ERROR GOTO 0 statement causes subsequent errors to transfer control to the BASIC error handler.

Examples

```
19000   ON ERROR GOTO 0
```

50.0 ON GOSUB

Function

The ON GOSUB statement transfers program control to one of several subroutines, depending on the value of a control expression.

Format

```
ON int-exp GOSUB target,... [ OTHERWISE target ]
```

Syntax Rules

1. *Target* must exist in the current program unit.
2. Control cannot be transferred into a statement block (such as FOR/NEXT, UNTIL/NEXT, WHILE/NEXT, DEF/END DEF, or SELECT/END SELECT).
3. You can use the ON GOSUB statement in a statement block if ON GOSUB and all its *targets* are inside that statement block.

General Rules

1. *Int-exp* determines which *target* BASIC selects as the GOSUB argument. If *int-exp* equals one, BASIC selects the first *target*. If *int-exp* equals two, BASIC selects the second *target*, and so on.
2. If there is an OTHERWISE clause, and if *int-exp* is less than one or greater than the number of *targets* in the list, BASIC selects the *target* of the OTHERWISE clause.
3. If there is no OTHERWISE clause, and if *int-exp* is less than one or greater than the number of *targets* in the list, BASIC signals "ON statement out of range" (ERR=58).
4. If a *target* specifies a nonexecutable statement, BASIC transfers control to the first executable statement that lexically follows the *target*.

Examples

```
150     ON CONTROL% GOSUB 100,200,300,400

200     ON A% GOSUB 10000,12000,14000 OTHERWISE 21000
```

51.0 ON GOTO

Function

The ON GOTO statement transfers program control to one of several lines, depending on the value of a control expression.

Format

```
                 { GO TO }
ON int-exp       { GOTO  }   target ,... [ OTHERWISE target ]
```

Syntax Rules

1. *Target* must exist in the current program unit.
2. Control cannot be transferred into a statement block (such as FOR/NEXT, UNTIL/NEXT, WHILE/NEXT, DEF/END DEF, SELECT/END SELECT).
3. You can use the ON GOTO statement in a statement block if ON GOTO and all its *targets* are inside that statement block.

General Rules

1. *Int-exp* determines which line number BASIC selects as the GOTO argument. If *int-exp* equals one, BASIC selects the first *target*. If *int-exp* equals two, BASIC selects the second *target*, and so on.
2. If there is an OTHERWISE clause, and if *int-exp* is less than one or greater than the number of targets in the list, BASIC transfers control to the *target* of the OTHERWISE clause.
3. If there is no OTHERWISE clause, and if *int-exp* is less than one or greater than the number of line numbers in the list, BASIC signals "ON statement out of range" (ERR=58).
4. If a *target* specifies a nonexecutable statement, BASIC transfers control to the first executable statement that lexically follows the *target*.

Examples

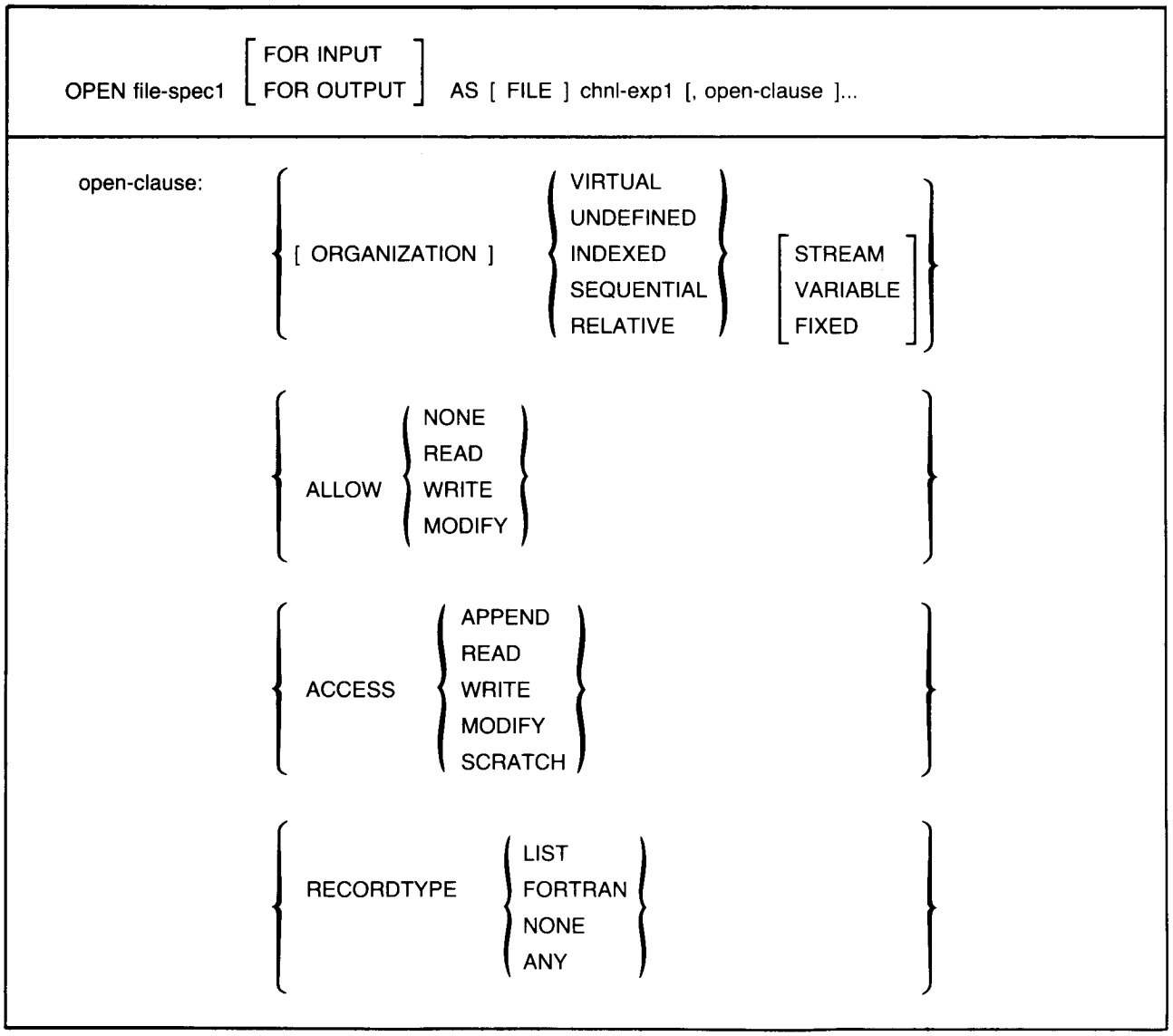
```
330     ON INDEX% GOTO 700,800,900 OTHERWISE 1000
```

52.0 OPEN

Function

The OPEN statement opens a file for processing. It transfers user-specified file characteristics to Record Management Services and verifies the results.

Format



(continued on next page)

```
{   RECORDSIZE int-exp1                                          }
{   FILESIZE int-exp2                                            }
{   WINDOWSIZE int-exp3                                          } (Except on RSTS/E)
{   TEMPORARY                                                    }
{   CONTIGUOUS                                                   }
{   MAP map-nam                                                  }
{   CONNECT chnl-exp2                                            }
{   BUFFER int-exp4                                              }
{   USEROPEN Func-nam                                            }
{   DEFAULTNAME file-spec2                                       }
{   EXTENDSIZE int-exp5                                          } (Except on RSTS/E)
{   MODE int-exp6                                                } (BASIC–PLUS–2 only)
{   CLUSTERSIZE int-exp7                                         } (BASIC–PLUS–2 on RSTS/E only)
{   BLOCKSIZE int-exp8                                           } (Sequential files)
{   NOREWIND                                                     } (Sequential files)
{   NOSPAN                                                       } (Sequential files)
{   SPAN                                                         } (Sequential files)
{   BUCKETSIZE int-exp9                                          } (Relative and Indexed files)
{   PRIMARY [ KEY ] key [ DUPLICATES ]                           } (Indexed files)
{   ALTERNATE [ KEY ] key [ DUPLICATES ] [ CHANGES ]             } (Indexed files)
{   UNLOCK EXPLICIT                                              } (VAX–11 BASIC only)

Key:  { str-unsubs-vbl
      { int-unsubs-vbl
      { decimal-unsubs-vbl
      { ( str-unsubs-vbl1 ,... str-unsubs-vbl8 )
```

Syntax Rules

1. *File-spec1* specifies the file to be opened and associated with *chnl-exp1*. It can be any valid string expression and must conform to your system's rules for file specifications. BASIC passes these values to RMS without editing, alteration, or validity checks.

   - *VAX–11 BASIC* does not supply any default file specifications unless you include the DEFAULTNAME clause in the OPEN statement.
   - *BASIC–PLUS–2* supplies the device as a default. If a device has been supplied in a previous OPEN statement, that device is used as the default. If there is no previous device, SY: is supplied as the default device. There is no default for the file type unless you include the DEFAULTNAME clause in the OPEN statement.

2. The FOR clause determines how BASIC opens a file.

   - If you open a file FOR INPUT, the file must exist or BASIC signals an error.
   - If you open a file FOR OUTPUT, BASIC creates the file if it does not exist. If the file does exist, *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* create a new version of the file. *BASIC–PLUS–2* on *RSTS/E systems* overwrites the existing file.
   - If you do not specify either FOR INPUT or FOR OUTPUT, BASIC tries to open an existing file. If there is no such file, BASIC creates one.

3. *Chnl-exp* is a numeric expression that specifies a channel number to be associated with *file-spec*. It can be preceded by an optional pound sign (#).

   - In *VAX–11 BASIC*, *chnl-exp* must be in the range 1 to 99.
   - In *BASIC–PLUS–2*, *chnl-exp* must be in the range 1 to 12.

4. The ORGANIZATION clause specifies the file organization. When present, it must precede all other clauses. When your OPEN statement has ORGANIZATION SEQUENTIAL, RELATIVE, or INDEXED, you get an RMS file.

   - On *VAX/VMS* and *RSX–11M/M–PLUS systems*, you get a terminal-format file when you omit the ORGANIZATION clause entirely. Terminal-format files are implemented as RMS sequential variable files and store ASCII characters in variable-length records. Carriage control is performed by the operating system; the record does not contain carriage returns or line feeds. You use essentially the same syntax to access terminal-format files as when reading from or writing to the terminal (INPUT and PRINT).
   - On *RSTS/E systems*, when you omit the ORGANIZATION clause, you get a terminal-format file that is a native mode RSTS/E ASCII stream file. RSTS/E ASCII stream files contain embedded carriage control characters. That is, carriage return and line feed characters are part of the record. See *BASIC on RSTS/E Systems* for more information on RSTS/E native mode files.

5. In the USEROPEN clause, *func-nam* must be a separately compiled FUNCTION subprogram and must conform to FUNCTION statement rules for naming subprograms.

6. The *key* specified in the PRIMARY KEY or ALTERNATE KEY clause must be declared in the MAP statement referenced by the OPEN statement.

General Rules

1. The OPEN statement does not retrieve records.
2. Channel zero, the terminal, is always open. If you try to open channel zero, *VAX–11 BASIC* signals the error "Illegal I/O channel" (ERR=46) and *BASIC–PLUS–2* signals "I/O channel already open at line <number>".
3. A statement that accesses a file cannot execute until you open that file and associate it with a *chnl-exp*.
4. If a program opens a file on a channel already associated with an open file, BASIC closes the previously opened file and opens the new one.
5. The FOR clause does not specify how your program can use the file or how others can share it. The ACCESS clause specifies how you use the file and the ALLOW clause specifies how the file is shared.
6. The ALLOW clause determines how other users can access the file:
   - ALLOW NONE lets no other users access the file. This is the default if any ACCESS other than READ is specified.
   - ALLOW READ lets other users have READ access to the file. This is the default for ACCESS READ.
   - ALLOW WRITE lets other users have WRITE access to the file.
   - ALLOW MODIFY lets other users have unlimited access to the file.
7. The ACCESS clause determines how the program can use the file:
   - ACCESS READ allows only FIND, GET, or other input statements on the file. The OPEN statement cannot create a file if the ACCESS READ clause is specified.
   - ACCESS WRITE allows only PUT, UPDATE, or other output statements on the file.
   - ACCESS MODIFY allows any I/O statement except SCRATCH on the file. ACCESS MODIFY is the default.
   - ACCESS SCRATCH allows any I/O statement valid for a sequential or terminal-format file.
   - ACCESS APPEND is the same as ACCESS WRITE for sequential files, except that BASIC positions the file pointer after the last record when it opens the file. You cannot use ACCESS APPEND on relative or indexed files.
8. The RECORDTYPE clause can be used only with RMS files. It specifies the file's record attributes:
   - LIST specifies implied carriage control, <CR> <LF> in *BASIC–PLUS–2*, and <CR> in *VAX–11 BASIC*. This is the default for all file organizations except VIRTUAL.
   - FORTRAN specifies a control character in the record's first byte.
   - NONE specifies no attributes. This is the default for VIRTUAL files.
   - ANY specifies a match with any file attributes when opening an existing file. If you create a new file, ANY is treated as LIST for all organizations except VIRTUAL. For VIRTUAL, it is treated as NONE.

9. The RECORDSIZE clause specifies the file's record size:

   - For ORGANIZATION FIXED, *int-exp1* specifies the size of all records.
   - For ORGANIZATION VARIABLE, *int-exp1* specifies the size of the largest record.
   - If you specify both a RECORDSIZE and a MAP clause, the RECORDSIZE clause overrides the record size set by the MAP clause. If you specify a MAP but no RECORDSIZE, the record size is equal to the MAP size. If there is no MAP, the RECORDSIZE clause determines the record size. If there is no MAP or RECORDSIZE specified, BASIC uses the default record size for the file organization when creating the file. When a program opens an existing file, BASIC uses the file's record size.
   - When creating SEQUENTIAL files, BASIC supplies a default record size of 132.
   - The record size is always 512 for VIRTUAL files unless you specify a RECORDSIZE.
   - If you do not specify a RECORDSIZE clause when opening an existing file, BASIC retrieves the record size value from the file. If you open a new file of ORGANIZATION RELATIVE and do not specify a RECORDSIZE clause, BASIC signals "Bad recordsize value on OPEN" (ERR = 148).

10. The FILESIZE clause lets you pre-extend a new file to a specified size. The value of *int-exp2* is the initial allocation of disk blocks. The FILESIZE clause has no effect on an existing file.

11. *Int-exp3* in the WINDOWSIZE clause lets you specify the number of block retrieval pointers you want to maintain in memory for the file. Retrieval pointers are associated with the file header and point to contiguous blocks on disk. By keeping retrieval pointers in memory, you can reduce the I/O associated with locating a record, as the operating system does not have to access the file header for pointers as frequently. The number of retrieval pointers in memory at any one time is determined by the system default or by the WINDOWSIZE clause. The usual default number of retrieval pointers on *RSX–11M/M–PLUS* and *VAX/VMS systems* is seven.

    - On *VAX/VMS systems*, a value of 0 specifies the default number of retrieval pointers. A value of 255 means to map the entire file, if possible. Values between 128 and 254, inclusive, are reserved.
    - On *RSX–11M/M–PLUS systems*, you can specify up to 127 retrieval pointers.
    - On *RSTS/E systems* the number of pointers in a window block is fixed at seven. Thus, you cannot use the WINDOWSIZE clause. You can, however, use the CLUSTERSIZE clause to increase the number of contiguous blocks mapped by one retrieval pointer.

12. The TEMPORARY clause causes BASIC to delete the output file as soon as the program closes it.

13. The CONTIGUOUS clause causes RMS to try to create the file as a contiguous sequence of disk blocks in *BASIC–PLUS–2* and as a contiguous-best-try sequence of disk blocks in *VAX–11 BASIC*. The CONTIGUOUS clause does not affect existing files or nondisk files.

14. The MAP clause specifies that a previously declared *map-nam* is associated with the file's record buffer. The MAP clause determines the record buffer's address and length unless overridden by the RECORDSIZE clause.

    - The size of the largest MAP with the same *map-nam* in the current program unit becomes the file's record size if the OPEN statement does not include a RECORDSIZE clause.
    - If there is no MAP clause, the record buffer space that BASIC allocates is not directly accessible. Thus, MOVE statements are needed to access data in the record buffer.
    - You must have a MAP clause when creating an indexed file; you cannot use KEY clauses without MAP statements since keys serve as offsets into the buffer.

15. The BUFFER clause can be used with all file organizations except UNDEFINED. For RELATIVE and INDEXED files, *int-exp4* specifies the number of device or file buffers Record Management Services uses for file processing. For SEQUENTIAL files, *int-exp4* specifies the size of the buffer; for example, BUFFER 8 for a SEQUENTIAL file sets the buffer size to eight 512–byte blocks.

16. The USEROPEN clause lets you open a file with your own FUNCTION subprogram. *Func-nam* must conform to the FUNCTION statement rules for naming subprograms. BASIC calls the user program after it fills the FAB (File Access Block), the RAB (Record Access Block), and the XABs (Extended Attribute Blocks). The subprogram must issue the appropriate RMS calls, including $OPEN and $CONNECT, and return the RMS status as the value of the function. See the *BASIC User's Guide* for more information on the USEROPEN routine.

17. The DEFAULTNAME clause lets you supply a default file specification. If *file-spec1* is not a complete file spec, *file-spec2* in the DEFAULTNAME clause supplies the missing parts. For example:

```
10       INPUT "FILE NAME";FNAM$
20       OPEN FNAM$ FOR INPUT AS FILE #1%, &
                    DEFAULTNAME "DB2:.DAT"
```

    If you type "ABC" for the file name, BASIC tries to open DB2:ABC.DAT. *BASIC–PLUS–2* allows DEFAULTNAME for RMS files only.

18. The EXTENDSIZE clause lets you specify the increment by which Record Management Services extends a file after its initial allocation is filled. The value of *int-exp5* is in 512–byte disk blocks.

19. The BLOCKSIZE clause specifies the physical blocksize of magnetic tape files. The value of *int-exp8* is the number of records in a block. Thus, the block size in bytes is the product of the RECORDSIZE and the BLOCKSIZE value. The default BLOCKSIZE is one record.

20. The NOREWIND clause controls tape positioning on magnetic tape files. If you specify neither ACCESS APPEND nor NOREWIND, the OPEN statement positions the tape at its beginning and then searches for the file.

21. The NOSPAN clause specifies that sequential records do not cross block boundaries. SPAN specifies that records can cross block boundaries. SPAN is the default. This clause does not affect nondisk files.

22. The BUCKETSIZE clause applies only to relative and indexed files. It specifies the size of an RMS bucket. The value of *int-exp9* is the number of records in a bucket. The default is one record.

23. The PRIMARY KEY clause lets you specify an indexed file's key. You must specify a PRIMARY KEY when opening an indexed file. The ALTERNATE KEY clause lets you specify up to 254 alternate keys. The ALTERNATE key clause is optional.

    - RMS creates one index list for each PRIMARY and ALTERNATE key you specify. These indexes are part of the file and contain pointers to the records. Each key you specify corresponds to a sorted list of record pointers.
    - The *keys* you specify determine the order in which records in the file are stored. All keys must be variables declared in the file's corresponding MAP statement. The position of the *key* in the MAP statement determines its position in the record. The data type and size of the *key* are as declared in the MAP statement.
    - A *key* can be an unsubscripted string or WORD variable in *BASIC–PLUS–2* and an unsubscripted string, WORD, LONG, or packed decimal variable in *VAX–11 BASIC*.
    - You can also create a segmented index key for string keys by separating the string variable names with commas and enclosing them in parentheses. You can then reference a segment of the specified key by referencing one of the string variables instead of the entire key. A string key can have up to eight segments.
    - The order of appearance of *keys* determines *key* numbers. The PRIMARY KEY, which must appear first, is key zero. The first ALTERNATE KEY is one, and so on.
    - DUPLICATES in the PRIMARY and ALTERNATE key clauses specifies that two records can have the same *key* value. If you do not specify DUPLICATES, the *key* value must be unique in all records.
    - CHANGES in the ALTERNATE key clause specifies that you can change the value of an alternate *key* when updating records. If you do not specify CHANGES when creating the file, you cannot change the value of a *key*. You cannot specify CHANGES with the PRIMARY KEY clause.

VAX–11 BASIC

1. If you open a terminal-format file with RECORDTYPE NONE, you must explicitly insert carriage control characters into the records your program writes to the file.

2. When you PRINT to a terminal-format file, you must supply a RECORDSIZE if the margin is to exceed 72 characters. For example, if you want to PRINT a 132–character line, specify RECORDSIZE 132 or use the MARGIN and NOMARGIN statements.

3. The CONTIGUOUS clause does not guarantee that the file will occupy a contiguous disk area. If RMS can locate the file in a contiguous area, it will do so. However, if there is not enough free contiguous space for a file, RMS allocates the largest possible contiguous areas and does not signal an error. See the *VAX–11 RMS User's Guide* for more information on contiguous disk allocation.

4. The CONNECT clause permits multiple record streams to be connected to the file.

   - The CONNECT clause must specify an INDEXED file already opened on *chnl-exp2* with the primary OPEN statement. You cannot connect to a connected channel, only to the initially opened channel. You can connect more than one stream to an open channel.
   - All clauses of the two files to be connected must be identical except MAP, CONNECT, and USEROPEN.

5. *VAX–11 RMS* does not allow the EXTENDSIZE clause for relative and indexed files.

6. If you specify NOREWIND, the OPEN statement does not position the tape. Your program can search for records from the current position.

7. The ALLOW clause can be used in the OPEN statement to specify file sharing of relative, indexed, sequential, and virtual files. But for sequential and virtual files, *VAX–11 RMS* restricts file sharing to files with fixed-length, 512–byte records. It does not allow the sharing of sequential files with variable-length records (the default), or of virtual files with recordsizes other than 512.

8. The UNLOCK EXPLICIT clause allows you to explicitly lock records with GET and FIND statements.

   - The type of lock you impose on a record with GET or FIND remains in effect until you explicitly unlock the record or file with a FREE or UNLOCK statement or until you close the file.
   - If you specify UNLOCK EXPLICIT, and do not impose a lock on a record with GET or FIND, BASIC imposes the ALLOW NONE lock by default and the next GET or FIND does not unlock the previously locked record.
   - You must open a file with UNLOCK EXPLICIT before you can lock records with GET and FIND statements. See the sections on GET and FIND in this manual and Chapter 8 in *BASIC on VAX/VMS Systems* for more information on explicit record locking and unlocking.

9. KEY clauses are optional for existing files if the keys in the file match BASIC defaults. If you do specify a *key*, it must match a *key* in the file.

BASIC–PLUS–2 (BP2)

1. The ORGANIZATION SEQUENTIAL STREAM clause specifies an RMS sequential stream file.
2. On *RSTS/E systems*, you can create both RMS sequential stream and *RSTS/E* ASCII stream files: (RSTS)

   - If you specify ORGANIZATION SEQUENTIAL STREAM, the file is RMS sequential stream.
   - If you omit the ORGANIZATION clause entirely, the file is *RSTS/E* ASCII (that is, a *RSTS/E* terminal-format file).

3. If you specify a CONTIGUOUS clause and there is not enough free contiguous space, RMS signals an error.

4. The CONNECT clause in *BASIC–PLUS–2* establishes additional record access streams for RMS files that allow your program to process more than one record of a file at the same time. Each stream represents an independent and concurrently active sequence of record operations.

   - The CONNECT clause must specify a RELATIVE or INDEXED file already open on *chnl-exp2*.
   - Each CONNECT established in a secondary OPEN statement uses another I/O channel. Because there are 12 I/O channels available, you can have a maximum of 12 connects to a file.
   - All clauses in the secondary OPEN statements must be identical except MAP, CONNECT, and USEROPEN.
   - *BASIC–PLUS–2* signals the error "Invalid file option" (ERR=139) if your program attempts to connect to a record stream that is already connected to another stream.

5. *BASIC–PLUS–2* provides the MODE clause for non–RMS file operations. *Int-exp6* specifies a MODE value.

   RSX
   - On *RSX–11M/M–PLUS systems,* MODE is ignored except when your program is doing device-specific I/O to a magnetic tape. In this case, you can use MODE to set the tape density. In all other cases, *RSX–11M/M–PLUS systems* ignore the MODE value. See *BASIC on RSX–11M/M–PLUS Systems* for information on MODE values.

   RSTS
   - On *RSTS/E systems,* MODE values affect only native-mode files, not RMS files. Further, MODE values have different meanings depending on the context in which you use them. This is because other pieces of software scan the MODE value to see which bits are set. For example, bit 14 may have one meaning to the *RSTS/E* terminal driver, another meaning to the file processor, and a third meaning to the diskette device driver. See *BASIC on RSTS/E Systems* for information on MODE values.

6. *BASIC–PLUS–2* on *RSTS/E systems* does not support the EXTENDSIZE clause.

7. On *RSTS/E systems,* you can specify the smallest amount of contiguous disk space to be allocated when an RMS or *RSTS/E* native-mode file's present allocation is exhausted. You do this with the CLUSTERSIZE clause. *Int-exp7* must be a power of two. For example, a CLUSTERSIZE of eight means that each time the file requires more disk space, the *RSTS/E operating system* must have at least eight contiguous disk blocks to allocate. If the disk is fragmented, there may be no eight-block clusters, and *BASIC–PLUS–2* signals the error "No room for user on device".

   - The default size of the clusters is a disk pack parameter set when the disk pack is initialized or mounted. This parameter, called a CLUSTER (of 512–byte blocks), becomes the default CLUSTERSIZE (the smallest amount of disk space that can be allocated for any file operations on that disk pack).
   - The CLUSTERSIZE clause does not affect the number of blocks read or written. It specifies only the smallest amount of disk space that can be allocated to a file.
   - *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* do not support the CLUSTERSIZE clause; however, the EXTENDSIZE clause serves a similar function.

8. If you specify NOREWIND, the OPEN FOR OUTPUT statement positions the tape at the logical end of the tape. The program can then write records. The OPEN FOR INPUT statement searches for the specified file without rewinding. If the file is not found, BASIC rewinds the tape and searches for the file from the start of the tape. If the file is still not found, BASIC signals the error "File not found".

Examples

```
100         OPEN "INPUT.DAT" FOR INPUT AS FILE #4,                  &
                ORGANIZATION SEQUENTIAL FIXED,                      &
                RECORDSIZE 200,                                     &
                MAP ABC,                                            &
                ALLOW MODIFY, ACCESS MODIFY

200         OPEN Newfile$ FOR OUTPUT AS FILE #3,                    &
                ORGANIZATION INDEXED,                               &
                MAP Emp_name,                                       &
                DEFAULTNAME "DB2:.DAT",                             &
                PRIMARY KEY Last$ DUPLICATES,                       &
                ALTERNATE KEY First$ CHANGES

100         MAP (SEGKEY) STRING LAST_NAME = 15,                     &
                FIRST_NAME = 15, MI = 1
200         OPEN "NAMES.IND" FOR OUTPUT AS FILE #1,                 &
                ORGANIZATION INDEXED,                               &
                PRIMARY KEY (LAST_NAME, FIRST_NAME, MI),            &
                MAP SEGKEY
```

53.0 OPTION

Function

The OPTION statement allows you to set compilation qualifiers such as default data type, size, and scale factor. In *VAX–11 BASIC*, you can also set compilation conditions such as subscript checking, overflow checking, decimal rounding, and setup in a source program. The defaults affect only the program module in which the OPTION statement occurs.

Format

```
OPTION option-clause,...

option-clause:    { TYPE = type-clause                                  }
                  { SIZE = size-clause                                  }
                  { SCALE = int-const                                   }
                  { { ACTIVE   } = active-clause   } (VAX–11 BASIC only)
                  { { INACTIVE }                   } (VAX–11 BASIC only)

type-clause:      { INTEGER   }
                  { REAL      }
                  { EXPLICIT  }
                  { DECIMAL   }  (VAX–11 BASIC only)

size-clause:      { size-item       }
                  { (size-item,...) }

size-item:        { INTEGER int-clause }
                  { REAL real-clause   }
                  { DECIMAL (d,s)      }  (VAX–11 BASIC only)

int-clause:       { BYTE }
                  { WORD }
                  { LONG }

real-clause:      { SINGLE  }
                  { DOUBLE  }
                  { GFLOAT  }  (VAX–11 BASIC only)
                  { HFLOAT  }  (VAX–11 BASIC only)

active-clause:    { active-item       }
                  { (active-item,...) }
```

active-item: { INTEGER OVERFLOW | DECIMAL OVERFLOW | SETUP | DECIMAL ROUNDING | SUBSCRIPT CHECKING } (all *VAX–11 BASIC* only)

Syntax Rules

1. *Option-clause* specifies the compilation qualifiers to be in effect for the program module.

   - *Type-clause* sets the default data type for variables not explicitly declared in the program module. You can specify only one *type-clause* in a program module.
   - *Size-clause* sets the default data subtypes for floating-point, integer, and (*VAX–11 BASIC* only) packed decimal data. *Size-item* specifies the data subtype you want to set. You can specify an INTEGER and/or REAL *size-item* in *BASIC–PLUS–2* and an INTEGER, REAL, and/or DECIMAL *size-item* in *VAX–11 BASIC*. Multiple *size-items* in an OPTION statement must be enclosed in parentheses and separated by commas.
   - SCALE controls the scaling of double precision floating-point variables. *Int-const* specifies the power of 10 you want as the scaling factor. It must be an integer from 0 to 6, inclusive, or BASIC signals an error. See the SCALE command in Section II of this manual for more information on scaling.
   - In *VAX–11 BASIC*, *active-clause* specifies the decimal rounding, integer and decimal overflow checking, setup, and subscript checking conditions you want in effect for the program module. *Active-item* specifies the conditions you want to set. Multiple *active-items* in an OPTION statement must be enclosed in parentheses and separated by commas. VAX

2. You can have more than one option in an OPTION statement, or you can use multiple OPTION statements in a program module. However, each OPTION statement must lexically precede all other source code in the program module, with the exception of comment fields, REM, SUB, FUNCTION, and OPTION statements.

General Rules

1. OPTION statement specifications apply only to the program module in which the statement appears and affect all variables in the module, including SUB and FUNCTION parameters.
2. BASIC signals an error in the case of conflicting options. For example, you cannot specify more than one *type-clause* or SCALE factor in the same program unit.
3. If you do not specify a *type-clause* or a *subtype-clause*, BASIC uses the current environment default data types.
4. If you do not specify a scale factor, BASIC uses the current environment default scale factor.

VAX

5. In *VAX–11 BASIC*, ACTIVE specifies the conditions that are to be in effect for a particular program module. INACTIVE specifies the conditions that are not to be in effect for a particular program module. If a condition does not appear in an *active-clause*, *VAX–11 BASIC* uses the current environment default for the condition. See Table 16 in this manual for information on the INTEGER_OVERFLOW, DECIMAL_OVERFLOW, SETUP, DECIMAL_ROUNDING, and SUBSCRIPT_CHECKING compilation qualifiers. These qualifiers correspond to *active-clause* conditions (INTEGER OVERFLOW, DECIMAL OVERFLOW, SETUP, DECIMAL ROUNDING, and SUBSCRIPT CHECKING).

Examples

```
10      FUNCTION REAL DOUBLE MONTHLY_PAYMENT,                &
                (DOUBLE INTEREST_RATE,                       &
                  LONG   NO_OF_PAYMENTS,                     &
                  DOUBLE PRINCIPLE)
20      OPTION TYPE = REAL,                                  &
                SIZE = (REAL DOUBLE, INTEGER LONG),          &
                SCALE = 4
```

54.0 PRINT

Function

The PRINT statement transfers program data to a terminal or a terminal-format file.

Format

To the Controlling Terminal

```
PRINT [ output-list ]
```

To a Channel

```
PRINT chnl-exp [, output-list ]
```

```
output-list:    [ exp ] [ sep [ exp ] ]...  [ sep ]

sep:            { , }
                { ; }
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#). If you do not specify a *chnl-exp*, BASIC prints to the controlling terminal.
2. *Output-list* specifies the expressions to be printed and the print format to be used.
3. *Exp* can be any valid expression.
4. A *sep* character must separate each *exp*. The *sep* characters control the print format:
   - A comma tells BASIC to skip to the next print zone before printing the expression.
   - A semicolon tells BASIC to print the expression immediately after the previous expression.

General Rules

1. A terminal-format file must be open on the specified *chnl-exp*.
2. A PRINT line has an integral number of print zones. Note, though, that the number of print zones in a line differs from terminal to terminal.
3. The right margin setting, if set by the MARGIN statement, controls the width of the PRINT line.
4. The PRINT statement prints string constants and variables exactly as they appear, with no leading or trailing spaces.

5. BASIC prints quoted string literals exactly as they appear. Thus, you can print quotation marks, commas, and other characters by enclosing them in quotation marks.

6. A PRINT statement with no *output-list* prints a blank line.

7. An *exp* in the *output-list* can be followed by more than one *sep* character. That is, you can omit an *exp* and specify where the next *exp* is to be printed by the use of multiple *sep* characters. For example:

```
100      PRINT "Name",,"Address and ";"City"
Run
PROGA           16-MAR-83 16:16

Name                            Address and City
```

   In this example, the double commas after "Name" cause BASIC to skip two print zones before printing "Address and ". The semicolon causes the next expression, "City", to be printed immediately after the preceding expression.

8. When printing numeric fields, BASIC precedes each number with a space or minus sign and follows it with a space. If a number can be represented exactly by six decimal digits or less, and, optionally, a decimal point, BASIC prints it that way.

9. BASIC rounds a number with an integer portion of six decimal digits or less (for example, 1234.567) to six digits (1234.57). If a number has more than six decimal digits, BASIC rounds the number to six digits and prints it in E format.

10. BASIC does not print trailing zeros to the right of the decimal point. If all digits to the right of the decimal point are zeros, BASIC omits the decimal point as well.

11. BASIC does not print more than six digits in explicit notation. If a number requires more than six digits, BASIC uses E format and precedes positive exponents with a plus sign (+).

12. The PRINT statement can print up to:

    - Three digits of precision for BYTE integers
    - Five digits of precision for WORD integers
    - Six digits of precision for SINGLE floating-point numbers
    - Ten digits of precision for LONG integers
    - Sixteen digits of precision for DOUBLE floating-point numbers
    - Fifteen digits of precision for GFLOAT floating-point numbers (*VAX–11 BASIC* only)
    - Thirty-three digits of precision for HFLOAT floating-point numbers (*VAX–11 BASIC* only)
    - Thirty-one digits of precision for DECIMAL numbers (*VAX–11 BASIC* only)
    - The string length for STRING values

13. A comma or semicolon can also follow the last item in *output-list*:

    - When printing to a terminal, BASIC does not generate a line terminator after printing the last item. The next item printed with a PRINT statement is printed at the position specified by the *sep* character following the last item in the first PRINT statement.
    - When printing to a terminal-format file, BASIC does not write out the record until a PRINT statement without trailing punctuation executes.

14. If no punctuation follows the last item in the *output-list*:

    - When printing to a terminal, BASIC generates a line terminator after printing the last item.
    - When printing to a terminal-format file, BASIC writes out the record after printing the last item.

15. If a string field does not fit on the current line:

    - When printing string elements to a terminal, BASIC prints as much as will fit on the current line and prints the remainder on the next line.
    - When printing string elements to a terminal-format file, BASIC prints the entire element on the next line.

16. If a numeric field is the first field in a line, and the numeric field spans more than one line, BASIC prints part of the number on one line and the remainder on the next. Otherwise, numeric fields are never split across lines. If the entire field cannot be printed at the end of one line, the number is printed on the next line.

17. When a number's trailing space does not fit in the last print zone, the number is printed without the trailing space.

18. *VAX–11 BASIC* rounds a floating point number with a magnitude between 0.1 and 1.0 to six digits. For magnitudes smaller than 0.1, BASIC rounds the number to six digits and prints it in E format.

19. For magnitudes smaller than 1, *BASIC–PLUS–2* prints up to five leading zeros and six significant digits in explicit point unscaled notation.

Examples

```
100    PRINT "THE ANSWER IS ";SUM%

200    PRINT #1, EMP_NUM, EMP_NAME$; EMP_AGE%
```

55.0 PRINT USING

Function

The PRINT USING statement generates output formatted according to a format string (either numeric or string) to a terminal or a terminal-format file.

Format

```
PRINT [ chnl-exp ] USING str-exp sep output-list

sep:          { , }
              { ; }

output-list:  [ exp ] [ sep [ exp ] ]... [ sep ]
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#). If you do not specify a *chnl-exp*, BASIC prints to the controlling terminal.
2. *Str-exp* is the format string. It must contain at least one valid format field and must be followed by a *sep* character and at least one *exp*.
3. *Output-list* specifies the expressions to be printed.
   - *Exp* can be any valid expression.
   - A *sep* character must separate each *exp*.
4. The *sep* characters in the PRINT USING statement do not control the print format as in the PRINT statement.

General Rules

1. The PRINT USING statement can print up to:
   - Three digits of precision for BYTE integers
   - Five digits of precision for WORD integers
   - Six digits of precision for SINGLE floating-point numbers
   - Ten digits of precision for LONG integers
   - Sixteen digits of precision for DOUBLE floating-point numbers
   - Fifteen digits of precision for GFLOAT floating-point numbers (*VAX–11 BASIC* only)
   - Thirty-three digits of precision for HFLOAT floating-point numbers (*VAX–11 BASIC* only)
   - Thirty-one digits of precision for DECIMAL numbers (*VAX–11 BASIC* only)
   - The string length for STRING values

2. A terminal-format file must be open on the specified *chnl-exp* or BASIC signals an error.
3. PRINT USING rounds a floating-point number once.
4. Format string characters control the format of numeric output.

   - The pound sign (#) reserves space for one sign or digit.
   - The comma (,) causes BASIC to insert commas before every third significant digit to the left of the decimal point. In the format field, the comma must be to the left of the decimal point, and to the right of the rightmost dollar sign, asterisk, or pound sign. A comma reserves space for a comma or digit.
   - The period (.) inserts a decimal point. The number of reserved places on either side of the period determines where the decimal point appears in the output.
   - The hyphen (–) reserves space for a sign and specifies trailing minus sign format. If present, it must be the last character in the format field. It makes BASIC print negative numbers with a minus sign after the last digit, and positive numbers with a trailing space. The trailing minus sign format (–) can be used as part of a dollar sign ($$) format field.
   - The letters CD enclosed in angle brackets (<CD>) print CR (Credit Record) after negative numbers or zero and DR (Debit Record) after positive numbers. If present, it must be the last format in the format field. The (<CD>) format can be used as part of a dollar sign ($$) format field.
   - Four carets(^^^^) specify E notation for floating-point numbers. They reserve four places for SINGLE, DOUBLE, and *VAX–11 BASIC* GFLOAT values and five places for *VAX–11 BASIC* HFLOAT values. If present, they must be the last characters in the format field.
   - Two dollar signs ($$) reserve space for a dollar sign and a digit and cause BASIC to print a dollar sign immediately to the left of the most significant digit.
   - Two asterisks (**) reserve space for two digits and cause BASIC to fill the left side of the numeric field with leading asterisks.
   - A zero enclosed in angle brackets (<0>) prints leading zeros instead of leading spaces.
   - A percent sign enclosed in angle brackets (<%>) prints all spaces in the field if the value of the print item is zero.

     Note

     When the dollar sign ($$), asterisk-fill (**), or zero-fill (<0>) formats are used to form one print field, they are mutually exclusive. Additionally, when the zero-fill (<0>) or blank-if-zero (<%>) formats are used to form one print field, they also are mutually exclusive.

   - An underscore (_) forces the next formatting character in the format string to be interpreted as a literal. It affects only the next character. If the next character is not a valid formatting character, the underscore has no effect and will itself be printed as a literal.

5. BASIC interprets any other characters in a numeric format string as string literals.

6. Depending on usage, the same format string characters can be combined to form one or more print fields within a format string. For example:

   - When a dollar sign ($$) or asterisk-fill (**) format precedes a pound sign (#) , it modifies the pound sign format. The ($$) or (**) reserves two places, and with the pound signs forms one print field. For example:

**$$###	forms one field and reserves five spaces
****##	forms one field and reserves four spaces

     When these formats are not followed by a pound sign or a blank-if-zero (<%>) format, they reserve two places and form a separate print field.

   - When a zero-fill (<0>) or blank-if-zero (<%>) format precedes a pound sign (#), it modifies the pound sign format. The (<0>) or (<%>) reserves one place, and with the pound signs forms one print field. For example:

**<0>####	forms one field and reserves five spaces
**<%>###	forms one field and reserves four spaces

     When these formats are not followed by a pound sign, they reserve one space and form a separate print field.

   - When a blank-if-zero (<%>) format follows a dollar sign ($$) or asterisk-fill (**) format, it modifies the ($$) or (**) format string. The (<%>) reserves one space, and with the ($$) or (**) format string forms one print field. For example:

**$$<%>###	forms one field and reserves six spaces
****<%>##	forms one field and reserves five spaces

     When the (<%>) precedes a ($$) or (**), it reserves one space and forms a separate print field.

7. In *VAX–11 BASIC*, the comma (digit separator), dollar sign (currency symbol), and decimal point (radix point) are the defaults for U.S. currency. The PRINT USING statement accesses the system-wide logical names for these symbols. To cause PRINT USING to format foreign currency, these logical names must be changed.

8. For E notation, PRINT USING left-justifies the number in the format field and adjusts the exponent to compensate, except when printing zero. When printing zero in E notation, BASIC prints leading spaces, leading zeros, a decimal point, and zeros in the fractional portion if the PRINT USING string contains these formatting characters, and then the string "E+00".

9. Zero cannot be negative. That is, if a small negative number rounds to zero, it is represented as a positive zero.

10. If there are reserved positions to the left of the decimal point, and the printed number is less than one, BASIC prints one zero to the left of the decimal point and pads with spaces to the left of the zero.

11. If there are more reserved positions to the right of the decimal point than fractional digits, BASIC prints trailing zeros in those positions.

12. If there are fewer reserved positions to the right of the decimal point than fractional digits, BASIC rounds the number to fit the reserved positions.

13. If a number does not fit in the specified format field, BASIC prints "%", followed by the number in PRINT format.

14. Format string characters control string output. All format characters except the backslash and exclamation point must start with a single quote ('). A single quote by itself reserves one character position. A single quote followed by format character(s) marks the beginning of a character format field and reserves one character position.

Note

VAX–11 BASIC accepts either upper– or lowercase string formatting characters. *BASIC–PLUS–2* accepts only uppercase string formatting characters.

- L reserves one character position. The number of Ls plus the leading single quote determines the field's size. BASIC left-justifies the print expression and pads with spaces if the print expression is less than or equal to the field's width. If the print expression is larger than the field, BASIC left-justifies the expression and truncates its right side to fit the field.

- R reserves one character position. The number of Rs plus the leading single quote determines the field's size. BASIC right-justifies the print expression and pads with spaces if the print expression is less than or equal to the field's width. If the print expression is larger than the field, BASIC left-justifies the expression and truncates its right side to fit the field.

- C reserves one character position. The number of Cs plus the leading single quote determines the field's size. If the string does not fit in the field, BASIC truncates its right side. Otherwise, BASIC centers the print expression in this field. If the string cannot be centered exactly, it is offset one character to the left.

- E reserves one character position. The number of Es plus the leading single quote determines the field's size. BASIC left-justifies the print expression if it is less than or equal to the field's width and pads with spaces. Otherwise, BASIC expands the field to hold the entire print expression.

- Two backslashes (\ \) when separated by n spaces reserve n+2 character positions. PRINT USING left-justifies the string in this field. BASIC does not allow a leading quotation mark with this format.

- An exclamation point (!) creates a 1–character field. The exclamation point both starts and ends the field. BASIC does not allow a leading quotation mark with this format.

Note

The backslash and exclamation formatting characters are included for compatibility with BASIC–PLUS. DIGITAL recommends that you do not use this type of character field for new program development.

15. BASIC interprets any other characters in the format string as string literals and prints them exactly as they appear.

16. A comma or semicolon can also follow the last item in *output-list*:

    - When printing to a terminal, BASIC does not generate a line terminator after printing the last item. The next item printed with a PRINT statement is printed at the position specified by the *sep* character following the last item in the first PRINT statement.
    - When printing to a terminal-format file, BASIC does not write out the record until a PRINT statement without trailing punctuation executes.

17. If no punctuation follows the last item in the *output-list*:

    - When printing to a terminal, BASIC generates a line terminator after printing the last item.
    - When printing to a terminal-format file, BASIC writes out the record after printing the last item.

Examples

```
500     PRINT USING "$$#####.##-", 8832.33, -88.3, A_VARIABLE

300     PRINT #1 USING "'E"; "NOW IS THE TIME FOR ALL GOOD MEN"
```

56.0 PUT

Function

The PUT statement transfers data from the record buffer to a file. PUT statements are valid on RMS sequential, relative, indexed, and block I/O files. You cannot use PUT statements on terminal-format files, virtual array files, or files opened with ORGANIZATION UNDEFINED.

Format

```
PUT chnl-exp [ , RECORD num-exp ] [ , COUNT int-exp ]
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. PUT with no RECORD clause writes data to an RMS sequential, relative, indexed, or block I/O file.
   - For sequential files, PUT adds a record at the end of the file.
   - For relative and block I/O files, PUT places the record in the empty cell pointed to by the Next Record Pointer. If the file is empty, the first PUT places a record in cell number one, the second in cell number two, and so on.
   - For indexed files, RMS stores records in order of ascending primary key value and updates all index keys.
3. The RECORD clause allows you to randomly write records to a relative or block I/O file by specifying the record number. *Int-exp* must be between one and the maximum record number defined in the OPEN statement.
4. *Int-exp* in the COUNT clause specifies the record's size. If there is no COUNT clause, the record's size is that defined by the MAP or RECORDSIZE clause (RECORDSIZE overrides MAP) in the OPEN statement.
   - If you write a record to a file with variable-length records, *int-exp* must be between zero and the maximum record size specified in the OPEN statement, inclusive.
   - If you write a record to a file with fixed-length records, the COUNT clause serves no purpose. If used, *int-exp* must equal the record size specified in the OPEN statement.
   - In *BASIC–PLUS–2*, if *int-exp* equals zero, the entire record is written to the file.

General Rules

1. For sequential access, the file associated with *chnl-exp* must be open with ACCESS WRITE, MODIFY, SCRATCH, or APPEND.
2. To add records to an existing sequential file, open it with ACCESS APPEND. If you are not at the end of the file when attempting a PUT to a sequential file, BASIC signals "Not at end of file" (ERR = 149).

3. For random access, the relative or block I/O file associated with *chnl-exp* must be open with ACCESS WRITE or MODIFY.

4. After a PUT statement executes, there is no Current Record Pointer. The Next Record Pointer is set as follows:

   - For sequential files, PUT sets the Next Record Pointer to the end-of-file.
   - For relative and block I/O files, a sequential PUT sets the Next Record Pointer to the Next Record plus one. A random PUT leaves the Next Record Pointer unchanged.
   - For indexed files, PUT leaves the Next Record Pointer unchanged.

5. When you specify a RECORD clause, BASIC evaluates *int-exp* and uses this value as the relative record number of the target cell.

   - If the target cell is empty or occupied by a deleted record, BASIC places the record in that cell.
   - If there is a record in the target cell, the PUT statement fails, and BASIC signals the error "Record already exists" (ERR = 153).

6. If an existing record in an indexed file has a record with the same key value as the one you want to PUT to the file, BASIC signals the error "Duplicate key detected" (ERR = 134) if you did not specify DUPLICATES for the key in the OPEN statement. If you specified DUPLICATES, RMS stores the records in a first-in, first-out sequence.

7. The number specified in the COUNT clause determines how many bytes are transferred from the buffer to a file:

   - If you have not completely filled the record buffer before executing a PUT, BASIC pads the record with nulls to equal the specified value.
   - If the specified COUNT value is less than the buffer size, the record is truncated to equal the specified value

8. The number in the COUNT clause must not exceed the size specified in the MAP or RECORDSIZE clause in the OPEN statement or BASIC signals "Size of record invalid" (ERR = 156).

9. Although block I/O files are implemented through RMS on *VAX/VMS systems* and *RSX–11M/M–PLUS systems*, when you write a record to a block I/O file, RMS does not perform the same error checking as with relative files. A PUT will write a record to a disk block specified in the RECORD clause, regardless of whether the block already contains a record. See Chapter 9 in the *BASIC User's Guide* for more information on RMS block I/O files. See the *RSTS/E Programming Manual* for information on RSTS/E native-mode block I/O files.

Examples

Sequential, Relative, Indexed, and Block I/O Files

```
700       PUT #3, COUNT 55%
```

Relative and Block I/O Files Only

```
2000      PUT #5, RECORD 133, COUNT 16%
```

57.0 RANDOMIZE

Function

The RANDOMIZE statement gives the random number function, RND, a new starting point.

Format

```
{ RANDOMIZE }
{ RANDOM    }
```

Syntax Rules

None.

General Rules

1. Without the RANDOMIZE statement, successive runs of the same program generate the same random number sequence. If you use the RANDOMIZE statement before invoking the RND function, the starting point changes for each run. Thus, a different random number sequence appears each time.

Examples

```
45      RANDOMIZE
```

58.0 READ

Function

The READ statement assigns values from a DATA statement to variables.

Format

```
READ vbl,...
```

Syntax Rules

1. In *VAX–11 BASIC*, *vbl* cannot be a DEF function name, unless the READ statement is inside the multi-line DEF body.
2. In *BASIC–PLUS–2*, *vbl* can be a DEF function name. If you assign a value to the DEF function name in this way, the next invocation of the function returns that value if it is not modified by the function body.
3. If your program has a READ statement without DATA statements, BASIC signals a compile-time error.

General Rules

1. When BASIC initializes a program unit, it forms a data sequence of all values in all DATA statements. An internal pointer points to the first value in the sequence.
2. When BASIC executes a READ statement, it sequentially assigns values from the data sequence to variables in the READ statement variable list. As BASIC assigns each value, it advances the internal pointer to the next value.
3. BASIC signals the error "Out of data" (ERR=57) if there are fewer data elements than READ statements. Extra data elements are ignored.
4. The data type of the value must agree with the data type of the variable to which it is assigned or BASIC signals "Data format error" (ERR=50).
5. IF you READ a string variable, and the DATA element is an unquoted string, BASIC ignores leading and trailing spaces. If the DATA element contains commas, they must be inside quotation marks.
6. BASIC evaluates subscript expressions in the variable list after it assigns a value to the preceding variable, and before it assigns a value to the subscripted variable. For example:

```
100       READ A, A$(A)
          !
          !
          !
500       DATA 10, NELSON
```

BASIC assigns the value 10 to variable A, then assigns the string "NELSON" to array element A$(10).

READ

Examples

```
1000      READ A, B%, C$
          !
          !
          !
2000      DATA 32.5, 8, ENDDATA
```

59.0 RECORD (VAX–11 BASIC)

Function

The RECORD statement lets you name and define data structures in a BASIC program and provides the BASIC interface to the VAX–11 Common Data Dictionary (CDD). You can use the defined RECORD name anywhere a BASIC data-type keyword is valid.

Format

```
RECORD rec-nam
        rec-component
              .
              .
              .

END RECORD [ rec-nam ]
```

```
rec-component:   { data-type rec-item [ , [ data-type ] rec-item ] }
                 { group-clause                                    }
                 { variant-clause                                  }

rec-item:        { unsubs-vbl [ = int-const ]                      }
                 { array ( int-const,... ) [ = int-const ]         }
                 { FILL [ ( int-const ) ] [ = int-const ]          }

group-clause:    GROUP group-nam [ ( int-const,... ) ]
                        rec-component
                              .
                              .
                              .
                 END GROUP [ group-nam ]

variant-clause:  VARIANT
                        case-clause
                              .
                              .
                              .
                 END VARIANT

case-clause:     CASE
                        [ rec-component ]
                              .
                              .
                              .
```

RECORD

Syntax Rules

1. Each line of text in a RECORD, GROUP, or VARIANT block can have an optional line number.

2. *Data-type* can be a BASIC data-type keyword or a previously defined *rec-nam* or *group-nam*. Table 2 lists and describes BASIC data-type keywords.

3. If the *data-type* of a *rec-item* is STRING, the string is fixed-length. You can supply an optional string length with the *=int-exp* clause. If you do not specify a string length, the default is 16.

4. *Int-const* in the *group-clause* specifies the number of times the GROUP block occurs in the RECORD data structure.

General Rules

1. *Rec-item* must conform to the rules for naming BASIC variables.

2. Variables and arrays in a record definition are also called elementary record components.

3. The RECORD statement names and defines a data structure called a record template, but does not allocate any storage. When you use the record template as a data type in a statement such as DECLARE, MAP, or COMMON, you declare a record instance. This declaration of the record instance allocates storage for the RECORD. For example:

```
1000    DECLARE EMPLOYEE EMP_REC
```

   This statement declares a variable named EMP_REC. EMP_REC is of the user-defined data type EMPLOYEE.

4. Whenever you access an elementary record component, that is, a variable named in a RECORD definition, you do it in the context of the record instance. Therefore, *rec-item* names need not be unique in your program. For example, you can have a variable called FIRST_NAME in any number of different RECORD definitions. However, you cannot use a BASIC keyword as a *rec-item* name and you cannot have two variables or arrays with the same name at the same level in the RECORD or GROUP definition.

5. The declarations between the RECORD statement and the END RECORD statement are called a RECORD block.

6. The declarations between the GROUP keyword and the END GROUP keywords are called a GROUP block. The GROUP keyword is valid only within a RECORD block.

7. A repeated GROUP is similar to an array within the record as it is zero-based. Thus, a repeat-count of 10 actually specifies 11 repetitions of the named GROUP.

8. The declarations between the VARIANT keyword and the END VARIANT keywords are called a VARIANT block.

9. The amount of space allocated for a VARIANT field in a RECORD is equal to the space needed for the variant field requiring the most storage. A record component outside of this overlaid field determines which record variant is used.

10. The *rec-nam* qualifies the *group-nam* and the *group-nam* qualifies the *rec-item*. You can access a particular *rec-item* within a record by specifying *rec-nam::group-nam::rec-item*. This specification is called a fully qualified reference. The full qualification of a *rec-item* is also called a component path name.

11. The *group-nam* is optional in a *rec-item* specification unless there is more than one *rec-item* with the same name. For example:

```
10      DECLARE EMPLOYEE EMP_REC
        !
        !
        !
100     RECORD Address
                STRING Street, City, State, Zip
        END RECORD Address
200     RECORD Employee
                GROUP Emp_name
                      STRING First = 15
                      STRING Middle = 1
                      STRING Last = 15
                END GROUP Emp_name
                ADDRESS Work
                ADDRESS Home
        END RECORD Employee
```

You can access the *rec-item* "Last" by specifying only "EMP_REC::Last" because only one *rec-item* is named "Last." However, if you try to reference "EMP_REC::City", BASIC signals an error because "City" is an ambiguous field, a component of both "Work" and "Home." To access "City," you must specify either "EMP_REC::Work::City" or "EMP_REC::Home::City."

Examples

```
1000      RECORD EMP_WAGE_CLASS
                  GROUP EMP_NAME
                          STRING Last = 15
                          STRING First = 14
                          STRING Middle = 1
                  END GROUP EMP_NAME
                  GROUP EMP_ADDRESS
                          STRING Street = 15
                          STRING City = 20
                          STRING State = 2
                          DECIMAL(5,0) Zip
                  END GROUP EMP_ADDRESS
                  STRING WAGE_CLASS = 2
                  VARIANT
                          CASE
                             GROUP HOURLY
                                DECIMAL(4,2) Hourly_wage
                                SINGLE Regular_pay_ytd
                                SINGLE Overtime_pay_ytd
                             END GROUP HOURLY
                          CASE
                             GROUP SALARIED
                                DECIMAL(7,2) Yearly_salary
                                SINGLE Pay_ytd
                             END GROUP SALARIED
                          CASE
                             GROUP EXECUTIVE
                                DECIMAL(8,2) Yearly_salary
                                SINGLE Pay_ytd
                                SINGLE Expenses_ytd
                             END GROUP EXECUTIVE
                  END VARIANT
          END RECORD EMP_WAGE_CLASS
```

60.0 REM

Function

The REM statement permits program documentation.

Format

```
REM [comment]
```

Syntax Rules

1. REM must be the only statement on the line or the last statement on a multi-statement line.
2. Because the REM statement is not executable, you can place it anywhere in a program, except where other statements, such as SUB and END SUB, must be the first or last statement in a program unit.
3. BASIC interprets every character between the keyword REM and the next line number as part of the comment.

General Rules

1. When the REM statement is the first statement on a line-numbered line, BASIC treats any reference to that line number as a reference to the next higher-numbered executable statement.
2. The REM statement is similar to the comment field that begins with an exclamation point, with one exception: the REM statement must be the last statement on a multi-statement line. The exclamation point comment field can be ended with a line terminator and followed by a BASIC statement. See Section I of this manual for more information on the comment field.

Examples

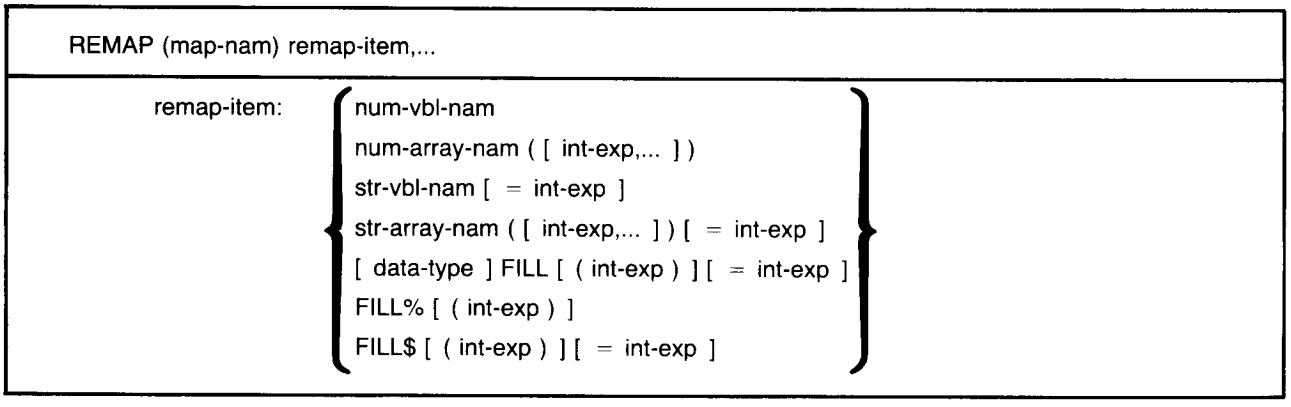
```
500     REM THIS IS A COMMENT
```

61.0 REMAP

Function

The REMAP statement defines or redefines the position in the record buffer of variables named in the MAP DYNAMIC statement.

Format



Syntax Rules

1. *Map-nam* is the name of a map area declared in the MAP and MAP DYNAMIC statements.

2. *Remap-item* names a variable, array, or array element declared in a preceding MAP DYNAMIC statement:

   - *Num-vbl-nam* specifies a numeric variable or array element. *Num-arr-nam* () specifies an entire numeric array.

   - *Str-vbl-nam* specifies a string variable or array element. *Str-arr-nam* () specifies an entire fixed-length string array. You can specify the number of bytes to be reserved for string variables and array elements with the =*int-exp* clause. The default string length is 16.

3. *Remap-item* can also be a FILL item. The FILL, FILL%, and FILL$ keywords let you reserve parts of the record buffer. *Int-exp* specifies the number of FILL items to be reserved. The =*int-exp* clause allows you to specify the number of bytes to be reserved for string FILL items. Table 21 describes FILL item format and storage allocation.

   Note

   In the applicable formats of FILL, *(int-const)* represents a repeat count, not an array subscript. FILL (n), for example, represents n elements, not n + 1.

4. All *remap-items*, except FILL items, must have been named in a previous MAP DYNAMIC statement, or BASIC signals an error.

5. *Data-type* can be any BASIC data-type keyword or, in *VAX–11 BASIC*, a data type defined in a RECORD statement. Data-type keywords, size, range, and precision are listed in Table 2 in this manual. You can specify a data type only for FILL items.

   - When you specify a *data-type*, all following FILL items are of that data type until you specify a new data type.
   - If you do not specify any *data-type*, FILL items take the current default data type and size.
   - FILL items following a *data-type* cannot end in a dollar sign or percent sign suffix character.

6. *Remap-items* must be separated with commas.

General Rules

1. The REMAP statement does not affect the amount of storage allocated to the map area.
2. Each time a REMAP statement executes, BASIC sets record pointers to the named map area for the specified variables from left to right.
3. The REMAP statement must be preceded by a MAP DYNAMIC statements or BASIC signals the error "No such MAP area <name>". The MAP statement creates a named area of static storage, the MAP DYNAMIC statement specifies the variables whose positions can change at run time, and the REMAP statement specifies the new positions for the variables names in the MAP DYNAMIC statement.
4. Until the REMAP statement executes, all variables named in the MAP DYNAMIC statement point to the first byte of the MAP area and all string variables have a length of zero. When the REMAP statement executes, BASIC sets the internal pointers as specified in the REMAP statement. For example:

```
100  MAP (DUMMY) STRING MAP_BUFFER = 50
     MAP DYNAMIC (DUMMY) LONG A, STRING B, SINGLE C(7)
     REMAP (DUMMY) B=14, A, C()
```

   The REMAP statement sets a pointer to byte 1 of DUMMY_MAP for string variable B, a pointer to byte 15 for LONG variable A, and pointers to bytes 19, 23, 27, 31, 35, 39, 43, and 47 for the elements in SINGLE array C.

5. You can use the REMAP statement to redefine the pointer for an array element or variable more than once in a single REMAP statement. For example:

```
100  MAP (DUMMY) STRING = 48
     MAP DYNAMIC (DUMMY) LONG A, B(10)
     REMAP (DUMMY) B(), B(0)
```

   This REMAP statement sets a pointer to byte 1 in DUMMY_MAP for array B. Since array B uses a total of 44 bytes, the pointer for the first element of array B, B(0) points to byte 45. References to array element B(0) will be to bytes 45 through 48. Pointers for array elements 1 through 10 are set to bytes 1, 4, 8, 12, and so forth.

6. Because the REMAP statement is local to a program module, it affects pointers only in the program module in which it executes.

Examples

```
100     MAP (EMPREC) STRING MAP_BUFFER = 100
        MAP DYNAMIC (EMPREC) STRING EMP_NAME,                    &
                             LONG BADGE_NO,                      &
                             STRING STREET, CITY, STATE,         &
                             WORD ZIP,                           &
                             STRING START_DATE
        REMAP (EMPREC) EMP_NAME = 20,                            &
                       BADGE_NO,                                 &
                       STREET = 10,                              &
                       CITY = 10,                                &
                       STATE = 2,                                &
                       ZIP,                                      &
                       START_DATE = 8
        !
        !
        !
500     REMAP (EMPREC) EMP_NAME = 10,                            &
                        BADGE_NO,                                &
                        STRING FILL = 32,                        &
                        WORD FILL,                               &
                        START_DATE = 8
```

63.0 RESUME

Function

The RESUME statement marks an exit point from an error-handling routine. BASIC clears the error condition and returns program control to a specified line number or to the program block in which the error occurred.

Format

```
RESUME [ lin-num ]
```

Syntax Rules

1. *Lin-num* must exist within the same program unit as the RESUME statement.
2. The RESUME statement cannot be used in a multi-line DEF unless the *lin-num* is also in the DEF function definition.

General Rules

1. The RESUME statement does not accept a label as an argument. Therefore, you should number lines that are to receive control from the error handler.
2. When no *lin-num* is specified in a RESUME statement, BASIC transfers control based on where the error occurs. If the error occurs on a numbered line containing a single statement, BASIC always transfers control to that statement. However, if the error occurs within a multi-statement line:
   - Within a FOR, WHILE, or UNTIL loop, BASIC transfers control to the first statement that follows the FOR, WHILE, or UNTIL statement.
   - Within a SELECT block, BASIC transfers control to the start of the CASE block in which the error occurs.
   - After a loop or SELECT block, BASIC transfers control to the statement that follows the NEXT or END SELECT statement.
   - If none of the above conditions occurs, BASIC transfers control back to the statement that follows the most recent line number or label.
3. To simplify and clarify error handling, DIGITAL recommends that the RESUME statement always be used with *lin-num*.

4. A RESUME statement with a specified *lin-num* transfers control to the first statement of a multi-statement line, regardless of which statement caused the error.

5. A RESUME statement cannot transfer control out of the current program unit. Thus, a RESUME statement with no *lin-num* cannot terminate an error handler in the following situation: (1) the error handler is handling an error that occurred in a subprogram or an external function, and (2) the error was passed to the calling program's error handler by an ON ERROR GO BACK statement or by default.

6. The execution of a RESUME with no *lin-num* is illegal if there is no error active. A RESUME with a *lin-num* is always legal. After clearing the error condition, BASIC transfers control to the specified line.

Examples

```
19100     RESUME 300

19990     RESUME
```

64.0 RETURN

Function

The RETURN statement transfers control to the statement immediately following the most recently executed GOSUB or ON GOSUB statement in the current program unit.

Format

```
RETURN
```

Syntax Rules

1. RETURN is the last statement executed in a subroutine even if it is not the last statement in the subroutine.

General Rules

1. Execution of a RETURN statement before the execution of a GOSUB or ON GOSUB causes BASIC to signal "RETURN without GOSUB" (ERR=72).

Examples

```
800     RETURN
```

65.0 RSET

Function

The RSET statement assigns right-justified data to a string variable. RSET does not change a string variable's length.

Format

```
RSET str-vbl,... = str-exp
```

Syntax Rules

1. BASIC evaluates the *str-vbl* subscript expression (if present) before assigning values.
2. *Str-vbl* cannot be a DEF function name, unless the RSET statement is inside the DEF function definition.

General Rules

1. The RSET statement treats strings as fixed-length. It does not change the length of *str-vbl* nor does it create new storage locations.
2. If *str-vbl* is longer than *str-exp*, RSET right-justifies the data and pads it with spaces on the left.
3. If *str-vbl* is shorter than *str-exp*, RSET truncates *str-exp* on the left.

Examples

```
100     RSET ZZ$ = "LMNOP"
```

66.0 SCRATCH

Function

The SCRATCH statement deletes the Current Record and all following records in an RMS sequential file.

Format

```
SCRATCH chnl-exp
```

Syntax Rules

None.

General Rules

1. Before you execute the SCRATCH statement, the file must be opened with ACCESS SCRATCH.
2. The SCRATCH statement applies to ORGANIZATION SEQUENTIAL files only.
3. The SCRATCH statement has no effect on terminals or unit record devices.
4. For disk files, the SCRATCH statement discards the current record and all that follow it in the file. The file is not physically shortened.
5. For magnetic tape files, the SCRATCH statement overwrites the current record with two end-of-file marks.

Examples

```
600     SCRATCH #4%
```

67.0 SELECT

Function

The SELECT statement lets you specify an expression, a number of possible values the expression may have, and a number of alternative statement blocks to be executed for each possible case. The END SELECT keywords terminate the SELECT block. The code between SELECT and END SELECT is called a SELECT block, and the code between CASE statements is called a CASE block.

Format

```
SELECT exp1
        case-clause
            .
            .
            .
        [ else-clause ]
END SELECT
```

```
case-clause:    CASE case-item,...
                        [ statement ] ...
case-item:      { [ rel-op ] exp2 }
                { exp3 TO exp4    }
else-clause:    CASE ELSE
                    [ statement ] ...
```

Syntax Rules

1. *Exp1* is the expression to be tested against the *case-clauses* and the *else-clause*. It can be numeric or string.

   - *Case-clause* consists of the CASE keyword followed by a *case-item* and statements to be executed when the *case-item* is true.
   - *Else-clause* consists of the CASE ELSE keywords followed by statements to be executed when no previous *case-item* has been selected as true.

2. *Case-item* is either an expression to be compared with *exp1* or a range of values separated with the keyword TO.

   - *Rel-op* is a relational operator specifying how *exp1* is to be compared to *exp2*. If you do not include a *rel-op*, BASIC assumes the equals (=) operator. BASIC executes the statements in the CASE block when the specified relational expression is true.

- *Exp3* and *exp4* specify a range of numeric or string values separated by the keyword TO. Separate multiple ranges with commas. BASIC executes the statements in the CASE block when *exp1* falls within any of the specified ranges.

3. A SELECT statement can have only one *else-clause*. The *else-clause* is optional and, when present, must be the last CASE block in the SELECT block.

General Rules

1. Each statement in a SELECT block can have its own line number.
2. The SELECT statement begins the SELECT BLOCK and the END SELECT keywords terminate it. BASIC signals an error if you do not include the END SELECT keywords.
3. Each CASE keyword establishes a CASE block. The next CASE or END SELECT keyword ends the CASE block.
4. You can nest SELECT blocks within a CASE or CASE ELSE block.
5. BASIC evaluates *exp1* when the SELECT statement is first encountered; BASIC then compares *exp1* with each *case-clause* in order of occurrence until a match is found or until a CASE ELSE block or END SELECT is encountered.
6. The following conditions constitute a match:
   - *Exp1* satisfies the relationship to *exp2* specified by *rel-op*.
   - *Exp1* is greater than or equal to *exp3*, but less than or equal to *exp4*, greater than or equal to *exp5* but less than or equal to *exp6*, and so on.
7. When a match is found between *exp1* and a *case-item*, BASIC executes the statements in the CASE block where the match occurred. If ranges overlap, the first match causes BASIC to execute the statements in the CASE block. After executing CASE block statements, control passes to the statement immediately following the END SELECT keywords.
8. If no CASE match occurs, BASIC executes the statements in the *else-clause*, if present, and then passes control to the statement immediately following the END SELECT keywords.
9. If no CASE match occurs and you do not supply a *case-else* clause, control passes to the statement following the END SELECT keywords.

Examples

```
100     SELECT A% + B% + C%
            CASE = 100
                    PRINT 'THE VALUE IS EXACTLY 100'
            CASE 1 TO 99
                    PRINT 'THE VALUE IS BETWEEN 1 AND 99'
            CASE > 100
                    PRINT 'THE VALUE IS GREATER THAN 100'
            CASE ELSE
                    PRINT 'THE VALUE IS LESS THAN 100'
        END SELECT
```

68.0 SLEEP

Function

The SLEEP statement suspends program execution for a specified number of seconds or until a carriage return is entered from the controlling terminal.

Format

```
SLEEP int-exp
```

Syntax Rules

1. In *VAX–11 BASIC*, *int-exp* must be between 0 and 2147483647, inclusive; if it is greater than 2147483647, BASIC signals the error "Integer error or overflow" (ERR=51).
2. In *BASIC–PLUS–2*, *int-exp* must be between 0 and 32767, inclusive; if it is greater than 32767, BASIC signals "Integer error" and does not suspend program execution.

General Rules

1. *Int-exp* is the number of seconds BASIC waits before resuming program execution.
2. Pressing the RETURN key on the controlling terminal cancels the effect of the SLEEP statement.

Examples

```
60    SLEEP 120%
```

62.0 RESTORE (RESET)

Function

The RESTORE statement resets the DATA pointer to the beginning of the DATA sequence or sets the record pointer to the first record in a file. RESET is a synonym for RESTORE.

Format

```
{ RESET   }
{ RESTORE }   [ chnl-exp [, KEY # int-exp ] ]
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).
2. *Int-exp* must be between zero and the number of keys in the file minus one, inclusive.

General Rules

1. The RESTORE statement is not allowed on virtual array files or on files opened on unit record devices.
2. If you do not specify a *chnl-exp*, RESTORE resets the DATA pointer to the beginning of the DATA sequence.
3. RESTORE affects only the current program unit. Thus, executing a RESTORE statement in a subprogram does not affect the DATA pointer in the main program.
4. If there is no *chnl-exp*, and the program has no DATA statements, RESTORE has no effect.
5. The file specified by *chnl-exp* must be open.
6. If *chnl-exp* specifies a magnetic tape file, BASIC rewinds the tape to the first record in the file.
7. The KEY clause applies to indexed files only. It sets a new key of reference equal to *int-exp* and sets the Next Record Pointer to the first logical record in that key.
8. For indexed files, the RESTORE statement without a KEY clause sets the Next Record Pointer to the first logical record specified by the current key of reference. If there is no current key of reference, the RESTORE statement sets the Next Record Pointer to the first logical record of the primary key.
9. If you use the RESTORE statement on any file type other than indexed, BASIC sets the Next Record Pointer to the first record in the file.

Examples

```
400     RESTORE #7%, KEY #4%
```

63.0 RESUME

Function

The RESUME statement marks an exit point from an error-handling routine. BASIC clears the error condition and returns program control to a specified line number or to the program block in which the error occurred.

Format

```
RESUME [ lin-num ]
```

Syntax Rules

1. *Lin-num* must exist within the same program unit as the RESUME statement.
2. The RESUME statement cannot be used in a multi-line DEF unless the *lin-num* is also in the DEF function definition.

General Rules

1. The RESUME statement does not accept a label as an argument. Therefore, you should number lines that are to receive control from the error handler.
2. When no *lin-num* is specified in a RESUME statement, BASIC transfers control based on where the error occurs. If the error occurs on a numbered line containing a single statement, BASIC always transfers control to that statement. However, if the error occurs within a multi-statement line:
   - Within a FOR, WHILE, or UNTIL loop, BASIC transfers control to the first statement that follows the FOR, WHILE, or UNTIL statement.
   - Within a SELECT block, BASIC transfers control to the start of the CASE block in which the error occurs.
   - After a loop or SELECT block, BASIC transfers control to the statement that follows the NEXT or END SELECT statement.
   - If none of the above conditions occurs, BASIC transfers control back to the statement that follows the most recent line number or label.
3. To simplify and clarify error handling, DIGITAL recommends that the RESUME statement always be used with *lin-num*.

4. A RESUME statement with a specified *lin-num* transfers control to the first statement of a multi-statement line, regardless of which statement caused the error.

5. A RESUME statement cannot transfer control out of the current program unit. Thus, a RESUME statement with no *lin-num* cannot terminate an error handler in the following situation: (1) the error handler is handling an error that occurred in a subprogram or an external function, and (2) the error was passed to the calling program's error handler by an ON ERROR GO BACK statement or by default.

6. The execution of a RESUME with no *lin-num* is illegal if there is no error active. A RESUME with a *lin-num* is always legal. After clearing the error condition, BASIC transfers control to the specified line.

Examples

```
19100      RESUME 300

19990      RESUME
```

64.0 RETURN

Function

The RETURN statement transfers control to the statement immediately following the most recently executed GOSUB or ON GOSUB statement in the current program unit.

Format

```
RETURN
```

Syntax Rules

1. RETURN is the last statement executed in a subroutine even if it is not the last statement in the subroutine.

General Rules

1. Execution of a RETURN statement before the execution of a GOSUB or ON GOSUB causes BASIC to signal "RETURN without GOSUB" (ERR=72).

Examples

```
800     RETURN
```

65.0 RSET

Function

The RSET statement assigns right-justified data to a string variable. RSET does not change a string variable's length.

Format

```
RSET str-vbl,... = str-exp
```

Syntax Rules

1. BASIC evaluates the *str-vbl* subscript expression (if present) before assigning values.
2. *Str-vbl* cannot be a DEF function name, unless the RSET statement is inside the DEF function definition.

General Rules

1. The RSET statement treats strings as fixed-length. It does not change the length of *str-vbl* nor does it create new storage locations.
2. If *str-vbl* is longer than *str-exp*, RSET right-justifies the data and pads it with spaces on the left.
3. If *str-vbl* is shorter than *str-exp*, RSET truncates *str-exp* on the left.

Examples

```
100     RSET ZZ$ = "LMNOP"
```

66.0 SCRATCH

Function

The SCRATCH statement deletes the Current Record and all following records in an RMS sequential file.

Format

```
SCRATCH chnl-exp
```

Syntax Rules

None.

General Rules

1. Before you execute the SCRATCH statement, the file must be opened with ACCESS SCRATCH.
2. The SCRATCH statement applies to ORGANIZATION SEQUENTIAL files only.
3. The SCRATCH statement has no effect on terminals or unit record devices.
4. For disk files, the SCRATCH statement discards the current record and all that follow it in the file. The file is not physically shortened.
5. For magnetic tape files, the SCRATCH statement overwrites the current record with two end-of-file marks.

Examples

```
600     SCRATCH #4%
```

67.0 SELECT

Function

The SELECT statement lets you specify an expression, a number of possible values the expression may have, and a number of alternative statement blocks to be executed for each possible case. The END SELECT keywords terminate the SELECT block. The code between SELECT and END SELECT is called a SELECT block, and the code between CASE statements is called a CASE block.

Format

```
SELECT exp1
        case-clause
            .
            .
            .
        [ else-clause ]
END SELECT
```

```
case-clause:    CASE case-item,...
                        [ statement ] ...
case-item:      { [ rel-op ] exp2 }
                { exp3 TO exp4    }
else-clause:    CASE ELSE
                    [ statement ] ...
```

Syntax Rules

1. *Exp1* is the expression to be tested against the *case-clauses* and the *else-clause*. It can be numeric or string.

   - *Case-clause* consists of the CASE keyword followed by a *case-item* and statements to be executed when the *case-item* is true.
   - *Else-clause* consists of the CASE ELSE keywords followed by statements to be executed when no previous *case-item* has been selected as true.

2. *Case-item* is either an expression to be compared with *exp1* or a range of values separated with the keyword TO.

   - *Rel-op* is a relational operator specifying how *exp1* is to be compared to *exp2*. If you do not include a *rel-op*, BASIC assumes the equals (=) operator. BASIC executes the statements in the CASE block when the specified relational expression is true.

- *Exp3* and *exp4* specify a range of numeric or string values separated by the keyword TO. Separate multiple ranges with commas. BASIC executes the statements in the CASE block when *exp1* falls within any of the specified ranges.

3. A SELECT statement can have only one *else-clause*. The *else-clause* is optional and, when present, must be the last CASE block in the SELECT block.

General Rules

1. Each statement in a SELECT block can have its own line number.
2. The SELECT statement begins the SELECT BLOCK and the END SELECT keywords terminate it. BASIC signals an error if you do not include the END SELECT keywords.
3. Each CASE keyword establishes a CASE block. The next CASE or END SELECT keyword ends the CASE block.
4. You can nest SELECT blocks within a CASE or CASE ELSE block.
5. BASIC evaluates *exp1* when the SELECT statement is first encountered; BASIC then compares *exp1* with each *case-clause* in order of occurrence until a match is found or until a CASE ELSE block or END SELECT is encountered.
6. The following conditions constitute a match:
   - *Exp1* satisfies the relationship to *exp2* specified by *rel-op*.
   - *Exp1* is greater than or equal to *exp3*, but less than or equal to *exp4*, greater than or equal to *exp5* but less than or equal to *exp6*, and so on.
7. When a match is found between *exp1* and a *case-item*, BASIC executes the statements in the CASE block where the match occurred. If ranges overlap, the first match causes BASIC to execute the statements in the CASE block. After executing CASE block statements, control passes to the statement immediately following the END SELECT keywords.
8. If no CASE match occurs, BASIC executes the statements in the *else-clause*, if present, and then passes control to the statement immediately following the END SELECT keywords.
9. If no CASE match occurs and you do not supply a *case-else* clause, control passes to the statement following the END SELECT keywords.

Examples

```
100     SELECT A% + B% + C%
                CASE = 100
                        PRINT 'THE VALUE IS EXACTLY 100'
                CASE 1 TO 99
                        PRINT 'THE VALUE IS BETWEEN 1 AND 99'
                CASE > 100
                        PRINT 'THE VALUE IS GREATER THAN 100'
                CASE ELSE
                        PRINT 'THE VALUE IS LESS THAN 100'
        END SELECT
```

68.0 SLEEP

Function

The SLEEP statement suspends program execution for a specified number of seconds or until a carriage return is entered from the controlling terminal.

Format

```
SLEEP int-exp
```

Syntax Rules

1. In *VAX–11 BASIC*, *int-exp* must be between 0 and 2147483647, inclusive; if it is greater than 2147483647, BASIC signals the error "Integer error or overflow" (ERR=51).
2. In *BASIC–PLUS–2*, *int-exp* must be between 0 and 32767, inclusive; if it is greater than 32767, BASIC signals "Integer error" and does not suspend program execution.

General Rules

1. *Int-exp* is the number of seconds BASIC waits before resuming program execution.
2. Pressing the RETURN key on the controlling terminal cancels the effect of the SLEEP statement.

Examples

```
60    SLEEP 120%
```

69.0 STOP

Function

The STOP statement halts program execution.

Format

```
STOP
```

Syntax Rules

None.

General Rules

1. STOP is valid anywhere in a program.
2. The STOP statement does not close files.

VAX–11 BASIC

1. When a STOP statement executes in a program executed with the RUN command in the BASIC environment, BASIC prints the line number and module name associated with the STOP statement, then displays the BASIC prompt. In response to the prompt, you can type immediate mode statements, CONTINUE to resume program execution, or any valid compiler command. See *BASIC on VAX/VMS Systems* for more information on immediate mode.
2. When a STOP statement is in an executable image, the line number, module name, and a pound sign (#) prompt are printed. In response to the prompt, you can type CONTINUE to continue program execution or EXIT to end the program. If the program module was compiled with the /NOLINE qualifier, no line number is displayed.

BASIC–PLUS–2

1. When a STOP statement executes in a program executed with the RUN/DEBUG command or compiled with the /DEBUG qualifier, control passes to the *BASIC–PLUS–2* debugger. The debugger prints the line number and module name associated with the STOP statement, then displays the pound sign (#) prompt. You can then use *BASIC–PLUS–2* debugger commands to analyze and debug your program. See Part VI in this manual for information on *BASIC–PLUS–2* debugger commands. Use the EXIT command to exit from the debugger and end the program.
2. When a STOP statement executes in a program executed with RUN or compiled without the /DEBUG qualifier, the line number of the STOP statement and a pound sign (#) prompt are printed. In response to the prompt, you can type CONTINUE to continue program execution or EXIT to end the program. The EXIT command closes all files before leaving the program.

Examples

```
95      STOP
```

70.0 SUB

Function

The SUB statement marks the beginning of a BASIC subprogram and specifies its parameters by number and data type.

Format

VAX–11 BASIC

VAX

```
SUB sub-name [ pass-mech ] [ ( [ formal-param ],... ) ]
        [ statement ]...
{ END SUB }
{ SUBEND  }

pass-mech: { BY DESC }
           { BY REF  }

formal-param: [ data-type ] { unsubs-vbl-nam                                          }
                            { array-nam ( [ int-const ] ,... ) } [ = int-const ] [ pass-mech ]
                                        [ ,           ] ...
```

BASIC–PLUS–2

BP2

```
SUB sub-name [ ( [ formal-param ],... ) ]
        [ statement ]...
{ END SUB }
{ SUBEND  }

formal-param: [ data-type ] { unsubs-vbl-nam                                          }
                            { array-nam ( [ int-const ] ,... ) }
                                        [ ,           ] ...
```

Syntax Rules

1. *Sub-nam* is the name of the separately compiled subprogram.
2. *Formal-param* specifies the number and type of parameters for the arguments the SUB subprogram expects to receive when invoked.
   - Empty parentheses indicate that the SUB subprogram has no parameters.
   - *Data-type* specifies the data type of a parameter. If you do not specify a data type, parameters are of the default data type and size. When you do specify a data type, all following parameters are of that data type until you specify a new data type. Data-type keywords, size, range, and precision are listed in Table 2 in this manual.
   - If you specify a datatype, *unsubs-vbl-nam* and *array-nam* cannot end in a percent sign (%) or dollar sign ($).
3. The SUB statement must be the first statement in the SUB subprogram.
4. Compiler directives and comment fields (!), because they are not BASIC statements, may precede the SUB statement. However, they cannot precede the subprogram's first numbered line. Note that REM is a BASIC statement; therefore, it cannot precede the SUB statement.
5. Every SUB statement must have a corresponding END SUB statement or SUBEND statement.
6. Any BASIC statement except END, FUNCTION, END FUNCTION, or EXIT FUNCTION can appear in a SUB subprogram.

VAX–11 BASIC

1. *Sub-nam* can consist of from 1 to 31 characters and must conform to the following rules:
   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), periods (.), or underscores (_).
   - A quoted name can consist of any combination of printable ASCII characters.
2. *Data-type* can be any BASIC data-type keyword or a data type defined in the RECORD statement.
3. If the data type is STRING, the *=int-const* clause allows you to specify the length of the string. If you do not specify a string length, the default length is 16.
4. *Pass-mech* specifies the parameter passing mechanism by which the subprogram receives arguments when called by non–BASIC programs.
5. A *pass-mech* clause outside the parentheses applies by default to all SUB parameters. A *pass-mech* clause in the *formal-param* list overrides the specified default and applies only to the immediately preceding parameter.
6. If you do not specify a *pass-mech*, the SUB program receives arguments by the default passing mechanisms, as shown in Table 19.

7. Parameters defined in *formal-param* must agree in number, type, ordinality, and *pass-mech* with the arguments specified in the CALL statement of the calling program.
8. You can specify from 1 to 32 *formal-params*.

BP2 *BASIC–PLUS–2*

1. *Sub-nam* can consist of from one to six characters and must conform to the following rules:
   - The first character of an unquoted name must be an alphabetic character (A through Z). The remaining characters, if present, can be any combination of letters, digits (0 through 9), dollar signs ($), or periods (.).
   - A quoted name can consist of any combination of alphabetic characters, digits, dollar signs ($), periods (.), or spaces.
2. *Data-type* can be any BASIC data-type keyword.
3. Parameters defined in *formal-param* must agree in number, type, and ordinality with the arguments specified in the CALL statement of the calling program.
4. You can specify from one to eight *formal-params*.

General Rules

1. All variables, except those named in MAP and COMMON statements and in DATA statements in a subprogram, are local to that subprogram.
2. BASIC initializes local variables upon each entry to the subprogram as follows:
   - Numeric variables are initialized to zero.
   - String variables are initialized to the null string.

VAX *VAX–11 BASIC*

1. SUB subprograms receive parameters BY REF or BY DESC. A SUB subprogram cannot receive any parameter BY VALUE. Table 19 lists and describes *VAX–11 BASIC* parameter passing mechanisms.
   - BY REF specifies that the subprogram receives the argument's address.
   - BY DESC specifies that the subprogram receives the address of a *VAX–11 BASIC* descriptor. For information about the format of a *VAX–11 BASIC* descriptor for strings and arrays, see Appendix C in *BASIC on VAX/VMS Systems*. For information on other types of descriptors, see the *VAX Architecture Handbook*.
2. By default, *VAX–11 BASIC* subprograms receive numeric *unsubs-vbls* BY REF and all other parameters BY DESC. You can override these defaults for strings and arrays with a BY clause:
   - If you specify a string length with the = *int-const* clause, you must also specify BY REF. If you specify BY REF and do not specify a string length, BASIC uses the default string length of 16.
   - If you specify array bounds, you must also specify BY REF.
3. RECORD data structures are initialized to zero or the null string.
4. *VAX–11 BASIC* subprograms may be called recursively.

BASIC–PLUS–2

BP2

1. You cannot specify how subprograms receive parameters in *BASIC–PLUS–2*. Numeric *unsubs-vbls* are received BY REF and string *unsubs-vbls* and entire arrays are received BY DESC. Table 20 lists and describes *BASIC–PLUS–2* BASIC parameter passing mechanisms.

   - BY REF specifies that the subprogram receives the argument's address.
   - BY DESC specifies that the subprogram receives the address of a *BASIC–PLUS–2* descriptor. For information about the format of a *BASIC–PLUS–2* descriptor, see Appendix C in *BASIC on RSX–11M/M–PLUS Systems* and *BASIC on RSTS/E Systems*.

2. *BASIC–PLUS–2* subprograms cannot be called recursively.

Examples

VAX–11 BASIC

```
100     SUB SUB3 BY REF (DOUBLE A, B,                          &
                         STRING EMP_NAM = 20 BY DESC,          &
                         WAGE(20))
        !
        !
        !
900     END SUB
```

BASIC–PLUS–2

```
100     SUB SUBPRO (BYTE AGE, DOUBLE WAGE(20,20), STRING EMP_NAME)
        !
        !
        !
900     END SUB
```

71.0 SUBEND

Function

The SUBEND statement is a synonym for END SUB. See the END statement for syntax rules.

Format

```
{ SUBEND  }
{ END SUB }
```

72.0 SUBEXIT

Function

The SUBEXIT statement is a synonym for the EXIT SUB statement. See the EXIT statement for syntax rules.

Format

```
{ SUBEXIT  }
{ EXIT SUB }
```

73.0 UNLESS

Function

UNLESS modifies a statement. BASIC executes the modified statement only if a conditional expression is false.

Format

```
statement UNLESS cond-exp
```

Syntax Rules

1. The UNLESS qualifier cannot be used on nonexecutable statements or on statements such as SELECT, IF, and DEF that establish a statement block.
2. *Cond-exp* can be any conditional expression.

General Rules

1. BASIC executes the statement only if *cond-exp* is false (value zero).

Examples

```
100     PRINT "A DOES NOT EQUAL 3" UNLESS A% = 3%
```

74.0 UNLOCK

Function

The UNLOCK statement unlocks the current record or bucket locked by the last FIND or GET statement.

Format

```
UNLOCK chnl-exp
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

General Rules

1. A file must be opened on *chnl-exp* before UNLOCK can execute.
2. The UNLOCK statement does not apply to files not on disk.
3. If the current record is not locked by a previous GET or FIND statement, the UNLOCK statement has no effect and BASIC does not signal an error.
4. The UNLOCK statement does not affect record buffers.
5. After you execute the UNLOCK statement, you cannot UPDATE or DELETE the current record.

Examples

```
90      UNLOCK #10
```

75.0 UNTIL

Function

The UNTIL statement marks the beginning of an UNTIL loop or modifies the execution of another statement.

Format

Conditional

```
UNTIL cond-exp
      [ statement ]...
NEXT
```

Statement Modifier

```
statement UNTIL cond-exp
```

Syntax Rules

1. *Cond-exp* can be any valid relational or logical expression.

Conditional

1. A NEXT statement must end the UNTIL loop.

General Rules

Conditional

1. BASIC evaluates *cond-exp* before each loop iteration. If the expression is false (value zero), BASIC executes the loop. If the expression is true (value nonzero), control passes to the first executable statement after the NEXT statement.

Statement Modifier

1. BASIC executes the statement repeatedly until *cond-exp* is true.

Examples

Conditional

```
10      UNTIL A >= 5
              A = A + .01
              TOTAL = TOTAL + 1
        NEXT
```

Statement Modifier

```
100      A = A + 1 UNTIL A >= 200
```

76.0 UPDATE

Function

The UPDATE statement replaces a record in a file with a record in the record buffer. UPDATE is valid only on RMS sequential, relative, and indexed files.

Format

```
UPDATE chnl-exp [ , COUNT int-exp ]
```

Syntax Rules

1. *Chnl-exp* is a numeric expression that specifies a channel number associated with a file. It must be immediately preceded by a pound sign (#).

2. *Int-exp* in the COUNT clause specifies the record's size.

3. In *BASIC–PLUS–2*, if *int-exp* equals zero, the entire record is written to the file.

General Rules

1. The file associated with *chnl-exp* must be a disk file opened with ACCESS MODIFY.

2. Each UPDATE statement must be preceded by a successful GET or FIND operation or BASIC signals "No current record" (ERR = 131). Because FIND locates but does not retrieve records, you must specify a COUNT clause in the UPDATE statement when the preceding operation was a FIND. *Int-exp* in the COUNT clause must exactly specify the size of the old record.

3. After an UPDATE statement executes, there is no Current Record Pointer. The Next Record Pointer is unchanged.

4. The length of the new record must be the same as that of the existing record for all files with fixed-length records. If the new record is larger than the existing record, BASIC truncates the right side of the new record to fit the existing record. If the new record is smaller than the existing record, the file gets corrupted.

5. If you write a record to a sequential file with fixed-length records, *int-exp* in the COUNT clause must exactly match the size of the old record.

6. For sequential files with variable-length records, the length of the new record must be the same as that of the existing record.

   - If you specify a COUNT clause, *int-exp* must match the size of the existing record.
   - In the absence of a COUNT clause, UPDATE uses the record size set by the last successful GET on that channel.

7. For relative files with variable-length records, the new record can be larger or smaller than the record it replaces.

   - The new record must be smaller than or equal to the maximum record size set with the MAP or RECORDSIZE clause when the file was opened.
   - You must use the COUNT clause to specify the size of the new record if it is different from that of the record last accessed by a GET on that channel.

8. For indexed files with variable-length records, the new record can be larger or smaller than the record it replaces.

   - When an indexed file permits duplicate primary keys, an updated record must be the same length as the old one.
   - When the program does not permit duplicate primary keys, the new record can be no longer than the maximum record size specified in the MAP or RECORDSIZE clause when the file was opened and must include at least the primary key field.
   - An alternate key for the new record can differ from that of the existing record only if the OPEN statement for that file specified CHANGES for the alternate key.

9. On *RSTS/E systems*, you can use UPDATE on native-mode files opened with mode 1 bit set (UPDATE mode).

Examples

```
100     UPDATE #4, COUNT 32
```

77.0 WAIT

Function

The WAIT statement specifies the number of seconds the program waits for terminal input before signaling an error.

Format

```
WAIT int-exp
```

Syntax Rules

1. The WAIT statement must precede an INPUT, INPUT LINE, LINPUT, MAT INPUT, or MAT LINPUT statement, or it has no effect.
2. In *VAX–11 BASIC*, *int-exp* must be between 0 and 2147483647, inclusive; if it is greater than 2147483647, BASIC signals the error "Integer error or overflow" (ERR = 51).
3. In *BASIC–PLUS–2*, *int-exp* must be between 0 and 32767, inclusive; if it is greater than 32767, BASIC signals "Integer error" and the WAIT statement has no effect.

General Rules

1. *Int-exp* is the number of seconds BASIC waits for input before signaling the error, "Keyboard wait exhausted" (ERR = 15).
2. After BASIC executes a WAIT statement, all input statements wait the specified amount of time before BASIC signals an error.
3. WAIT 0 disables the WAIT statement.

Examples

```
50      WAIT 60
        INPUT "YOU HAVE SIXTY SECONDS TO TYPE YOUR NAME"; NAME$
        WAIT 0
```

78.0 WHILE

Function

The WHILE statement marks the beginning of a WHILE loop or modifies the execution of another statement.

Format

Conditional

```
WHILE cond-exp
        [ statement ]...
NEXT
```

Statement Modifier

```
statement WHILE cond-exp
```

Syntax Rules

1. *Cond-exp* can be any valid relational or logical expression.

Conditional

1. A NEXT statement must end the WHILE loop.

General Rules

Conditional

1. BASIC evaluates *cond-exp* before each loop iteration. If the expression is true (value non-zero), BASIC executes the loop. If the expression is false (value zero), control passes to the first executable statement after the NEXT statement.

Statement Modifier

1. BASIC executes the statement repeatedly as long as *cond-exp* is true.

Examples

Conditional

```
10      WHILE X < 100
            X = X + SQR(X)
        NEXT
```

Statement Modifier

```
100     X% = X% + 1% WHILE X% < 100%
```

PART V
Functions

1.0 ABS

Function

The ABS function returns a floating-point number that equals the absolute value of a specified floating-point expression.

Format

```
real-vbl = ABS(real-exp)
```

Syntax Rules

None.

General Rules

1. BASIC expects the argument of the ABS function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

2. The returned floating-point value is always greater than or equal to zero. The absolute value of zero is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by minus one.

Examples

```
400     A = ABS(-100 * G)
410     B = -39.2
420     PRINT ABS(B), A
```

2.0 ABS%

Function

The ABS% function returns an integer number that equals the absolute value of a specified integer expression.

Format

```
int-vbl = ABS%(int-exp)
```

Syntax Rules

None.

General Rules

1. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default integer size.
2. The returned value is always greater than or equal to zero. The absolute value of zero is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by minus one.

Examples

```
400     A = ABS%(-100% * G%)
410     B = -39
420     PRINT ABS%(B), A
```

3.0 ASCII

Function

The ASCII function returns the ASCII value (base 10) of a string's first character.

Format

```
                 { ASC   }
int-vbl  =       { ASCII } (str-exp)
```

Syntax Rules

None.

General Rules

1. The ASCII value of a null string is zero.
2. The ASCII function returns an integer value of the default size between 0 and 255, inclusive.

Examples

```
500     ASC_VAL = ASCII(EMP_NAM$)
```

4.0 ATN

Function

The ATN function returns the angle, in radians, of a specified tangent.

Format

```
real-vbl = ATN(real-exp)
```

Syntax Rules

None.

General Rules

1. ATN returns a value from –PI/2 through PI/2.
2. The returned angle is expressed in radians.
3. BASIC expects the argument of the ATN function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

```
150     ANGLE_RAD = ATN(T)
160     ANGLE_DEG = ANGLE_RAD/(PI/180)
```

5.0 BUFSIZ

Function

The BUFSIZ function returns the buffer size, in bytes, of a specified channel.

Format

```
int-vbl = BUFSIZ(chnl-exp)
```

Syntax Rules

None.

General Rules

1. *Chnl-exp* is the channel expression of an open file. If the specified channel is closed, BUFSIZ returns zero. You cannot include a pound sign (#) in *chnl-exp*.
2. In *BASIC–PLUS–2*, BUFSIZ of channel zero returns the current terminal width or, in a batch stream, 512.
3. In *VAX–11 BASIC*, BUFSIZ of channel zero always returns 132.
4. *Int-vbl* is a WORD integer in *BASIC–PLUS–2* and a LONG integer in *VAX–11 BASIC*.

Examples

```
100     A = BUFSIZ(2)
```

6.0 CCPOS

Function

The CCPOS function returns the output record's current character or cursor position on a specified channel.

Format

```
int-vbl = CCPOS(chnl-exp)
```

Syntax Rules

None.

General Rules

1. *Chnl-exp* must specify an open file or terminal. You cannot include a pound sign (#) in *chnl-exp*.
2. If *chnl-exp* is zero, CCPOS returns the current character position of the controlling terminal.
3. The *int-vbl* returned by the CCPOS function is of the default integer size.
4. The CCPOS function counts only characters. If you use cursor addressing sequences such as escape sequences, the value returned will not be the cursor position.
5. The first character position on a line is zero.

Examples

```
100     CHNL0 = CCPOS (0)
```

7.0 CHR$

Function

The CHR$ function returns a 1–character string that corresponds to the ASCII value you specify.

Format

```
str-vbl = CHR$(int-exp)
```

Syntax Rules

None.

General Rules

1. CHR$ returns the character whose ASCII value equals *int-exp*. If *int-exp* is greater than 255, BASIC treats it modulo 256. For example, CHR$(325) is the same as CHR$(69).
2. BASIC treats all arguments as unsigned 8–bit integers in the range 0 to 255. Negative numbers are treated as the two's complement (for example, –1 is treated as 255).
3. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
220     A$ = CHR$(65)
230     PRINT CHR$(VALUE)
```

8.0 COMP%

Function

The COMP% function compares two numeric strings and returns a minus one, zero, or one, depending on the results of the comparison.

Format

```
int-vbl = COMP%(str-exp1, str-exp2)
```

Syntax Rules

1. *Str-exp1* and *str-exp2* are numeric strings. They can contain up to 60 ASCII digits and an optional decimal point and leading sign.

General Rules

1. If *str-exp1* is greater than *str-exp2*, COMP% returns one.
2. If the string expressions are equal, COMP% returns zero.
3. If *str-exp1* is less than *str-exp2*, COMP% returns minus one.
4. The value returned by the COMP% function is an integer of the default size.

Examples

```
400     NUM_STRING$ = "35"
425     OLD_NUM_STRING$ = "33.1"
450     ALPHA = COMP%(NUM_STRING$, OLD_NUM_STRING$)
```

9.0 COS

Function

The COS function returns the cosine, in radians, of an angle.

Format

```
real-vbl = COS(real-exp)
```

Syntax Rules

None.

General Rules

1. The returned value is between minus one and one.
2. BASIC expects the argument of the COS function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

```
900     COSINE.ALPHA = COS(PI/2)
```

10.0 CTRLC

Function

The CTRLC function enables CTRL/C trapping. When CTRL/C trapping is enabled, a CTRL/C typed at the terminal causes control to be transferred to the program's error handler.

Format

```
int-vbl = CTRLC
```

Syntax Rules

None.

General Rules

1. After the CTRLC function is invoked, control passes to the error handler when BASIC encounters a CTRL/C. If there is no error handler in a program, the program aborts when BASIC encounters a CTRL/C.
2. CTRL/C trapping is asynchronous; that is, BASIC suspends execution and signals "Programmable ^C trap" (ERR=28) as soon as it detects a CTRL/C. Consequently, a statement can be interrupted while executing. A statement so interrupted may be only partially completed and variables may be left in an undefined state.
3. BASIC can trap more than one CTRL/C error in a program as long as the error does not occur while the error handler is executing. If a second CTRL/C is detected while the error handler is processing the first CTRL/C, the program aborts.

RSX

4. On *RSX–11M/M–PLUS systems*, the task that contains the CTRLC function must be able to attach to a terminal as soon as the CTRLC function is enabled. If another task is attached to the terminal, the task that enabled the CTRLC function terminates with a directive error.
5. The CTRLC function always returns a value of zero.

Examples

```
10      ON ERROR GOTO 19000
20      Y% = CTRLC
          .
          .
          .
19000   IF (ERR = 28) THEN Y% = CTRLC
19010   RESUME
```

11.0 CVT$$

Function

The CVT$$ function is identical to the EDIT$ function. See the EDIT$ function for syntax and general rules.

Note

DIGITAL recommends that you use the EDIT$ function rather than the CVT$$ function for new program development.

Format

```
str-vbl = CVT$$(str-exp, int-exp)
```

Examples

```
100     A$ = CVT$$(B$,48)
```

12.0 CVTxx

Function

Note

CVT functions are supported only for compatibility with BASIC–PLUS. DIGITAL recommends that you use BASIC's dynamic mapping feature or multiple MAP statements for new program development.

The CVT$% function maps the first 2 characters of a string into a 16–bit integer. The CVT%$ function translates a 16–bit integer into a 2–character string. The CVT$F function maps a 4– or 8–character string into a floating-point variable. The CVTF$ function translates a floating-point number into a 4– or 8–byte character string. The number of characters translated depends on whether the floating-point variable is single- or double-precision.

Format

```
int-vbl = CVT$%(str-vbl)

str-vbl = CVT%$(int-vbl)

str-vbl = CVTF$(real-vbl)

real-vbl = CVT$F(str-vbl)
```

Syntax Rules

1. In *VAX–11 BASIC*, CVT functions reverse the order of the bytes when moving them to or from a string. Thus, you can mix MAP and MOVE statements, but you cannot use FIELD and CVT functions on a file if you also plan to use MAP or MOVE.

General Rules

CVT$%

1. If the CVT$% *str-vbl* has fewer than two characters, BASIC pads the string with nulls.

2. In *VAX–11 BASIC*, if the default data type is LONG, only two bytes of data are extracted from *str-vbl*; the high-order byte is sign-extended into a longword.
3. The value returned by the CVT$% function is an integer of the default size.

CVT%$

1. Only two bytes of data are inserted into *str-vbl*.
2. If you specify a floating-point variable for *int-vbl*, BASIC truncates it to an integer of the default size. If the default size is BYTE and the value of *int-vbl* exceeds 127, BASIC signals an error.

CVT$F

1. CVT$F maps four characters when the program is compiled with /SINGLE and eight characters when the program is compiled with /DOUBLE.
2. If *str-vbl* has fewer than four or eight characters, BASIC pads the string with nulls.
3. The *real-vbl* returned by the CVT$F function is of the default floating-point size. In *VAX–11 BASIC*, if the default size is GFLOAT or HFLOAT, BASIC signals the error "Floating CVT illegal for GFLOAT or HFLOAT".

CVTF$

1. The CVTF$ function maps single-precision numbers to a 4–character string and double-precision numbers to an 8–character string.
2. BASIC expects the argument of the CVTF$ function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size. In *VAX–11 BASIC*, if the default floating-point size is GFLOAT or HFLOAT, BASIC signals the error "Floating CVT illegal for GFLOAT or HFLOAT".

Examples

```
10      A% = CVT$%(EMP_NAME$)

20      A$ = CVT%$(A%)

100     A = CVT$F(EMP_NAME$)

110     EMP_NAME$ = CVTF$(A)
```

Note

DIGITAL does not recommend the CVTxx functions for new program development.

13.0 DATE$

Function

The DATE$ function returns a string containing a day, month, and year in the form dd–Mmm–yy.

Format

```
str-vbl = DATE$(int-exp)
```

Syntax Rules

1. *Int-exp* can have up to six digits in the form YYYDDD, where the "Y" characters specify the number of years since 1970 and the "D" characters specify the day of that year.
2. You must fill all three of the "D" positions with digits or zeros before you fill the "Y" positions. For example:
   - DATE$(121) returns the date 01–May–70, day 121 of the year 1970.
   - DATE$(1201) returns the date 20–Jul–71, day 201 of the year 1971.
   - DATE$(12001) returns the date 01–Jan–82, day 1 of the year 1982.
   - DATE$(10202) returns the date 21–Jul–80, day 202 of the year 1980.
3. If *int-exp* equals zero, DATE$ returns the current date.

General Rules

1. The *str-exp* returned by the DATE$ function consists of nine characters and expresses the day, month, and year in the form dd-Mmm-yy.
2. If you specify an invalid date, such as day 385, results are undefined and unpredictable.
3. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.
4. On *RSTS/E systems*, the form of the DATE$ function's output can be changed to ISO format, yy.mm.dd, during the installation procedure, or to the format selected by the system manager at system start-up time.

Examples

```
500     PRINT DATE(9231)
```

14.0 DECIMAL (VAX–11 BASIC Only)

Function

The DECIMAL function converts a numeric expression or numeric string to the DECIMAL data type.

Format

```
decimal-vbl = DECIMAL(exp [, int-const1, int-const2 ] )
```

Syntax Rules

1. *Int-const1* specifies the total number of digits (the precision) and *int-const2* specifies the number of digits to the right of the decimal point (the scale). If you do not specify these values, BASIC uses the d (digits) and s (scale) defaults for the DECIMAL data type.
2. *Int-const1* and *int-const2* must be positive integers in the range 1 to 31, inclusive. *Int-const2* cannot exceed the value of *int-const1*.
3. *Exp* can be either a numeric expression or a numeric string. If a numeric string, it can contain up to 31 ASCII digits and an optional decimal point and leading sign.

General Rules

1. If *exp* is a string, BASIC ignores leading, trailing, and embedded spaces and tabs.
2. The DECIMAL function returns a zero when a string argument contains only spaces and tabs, or when it is null.

Examples

```
100     INPUT "enter a decimal value";DEC_VALUE
            B = DECIMAL(DEC_VALUE,5,2)
            PRINT B, DECIMAL(HOURLY_PAY)
```

15.0 DET

Function

The DET function returns the value of the determinant of the last matrix inverted with the MAT INV function.

Format

```
real-vbl = DET
```

Syntax Rules

None.

General Rules

1. When a matrix is inverted with the MAT INV statement, BASIC calculates the determinant as a by-product of the inversion process. The DET function retrieves this value.
2. If your program does not contain a MAT INV statement, the DET function returns a zero.
3. The value returned by the DET function is a floating-point value of the default floating-point size.

Examples

```
100     DETERMINANT = DET
        PRINT DET
```

16.0 DIF$

Function

DIF$ returns a string whose value is the difference between two numeric strings.

Format

```
str-vbl = DIF$(str-exp1, str-exp2)
```

Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to process. They can contain up to 54 ASCII digits, an optional decimal point, and an optional leading sign.

General Rules

1. BASIC subtracts *str-exp2* from *str-exp1* and stores the result in *str-vbl*.
2. The difference between two integers takes the precision of the larger integer.
3. The difference between two decimal fractions takes the precision of the more precise fraction, unless trailing zeros generate that precision.
4. The difference between two floating-point numbers takes precision as follows:
   - The difference of the integer parts takes the precision of the larger part.
   - The difference of the decimal fraction part takes the precision of the more precise part.
5. BASIC truncates leading and trailing zeros.

Examples

```
500     RESULT$ = DIF$("6776", "-455")
```

17.0 ECHO

Function

The ECHO function causes characters to be echoed at a terminal open on a specified channel.

Format

```
int-vbl = ECHO(chnl-exp)
```

Syntax Rules

None.

General Rules

1. *Chnl-exp* must specify a terminal. You cannot include a pound sign (#) in *chnl-exp*.
2. The ECHO function is the complement of the NOECHO function; that is, ECHO disables the effect of ECHO and vice versa.
3. The ECHO function has no effect on an unopened channel.
4. The ECHO function always returns a value of zero.

Examples

```
100     Y = ECHO(0)
```

18.0 EDIT$

Function

The EDIT$ function performs one or more string editing functions, depending on the value of its integer argument.

Format

```
str-vbl = EDIT$(str-exp, int-exp)
```

Syntax Rules

None.

General Rules

1. BASIC edits *str-exp* to produce *str-vbl*.
2. The editing that BASIC performs depends on the value of *int-exp*. Table 22 describes EDIT$ values and functions.
3. All values are additive; that is, you can perform the editing functions of values 8, 16, and 32 by specifying a value of 56.
4. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Table 22: EDIT$ Values

Value	Edit Performed
1	Discards each character's parity bit (bit 7)
2	Discards all spaces and tabs
4	Discards all carriage returns, line feeds, form feeds, deletes, escapes, and nulls
8	Discards leading spaces and tabs
16	Converts multiple spaces and tabs to a single space
32	Converts lowercase letters to uppercase letters
64	Converts left bracket ([) to left parenthesis [(] and right bracket (]) to right parenthesis [)]
128	Discards trailing spaces and tabs (same as TRM$ function)
256	Suppresses all editing for characters within quotation marks; if the string has only one quotation mark, BASIC suppresses all editing for the characters following the quotation mark

Examples

```
100     NEW_STRING$ = EDIT$(OLD_STRING$, 32 + 16)
```

19.0 ERL

Function

The ERL function returns the number of the line where the last error occurred.

Format

```
int-vbl = ERL
```

Syntax Rules

None.

General Rules

1. If the ERL function is used before an error occurs or after BASIC executes a RESUME statement, results are undefined.
2. The ERL function overrides the /NOLINE qualifier. If a program compiled with the /NOLINE qualifier in effect contains an ERL function, BASIC signals the message "ERL overrides NOLINE".
3. The *int-vbl* returned by the ERL function is a WORD integer in *BASIC–PLUS–2* and a LONG integer in *VAX–11 BASIC*.

Examples

```
300     IF (ERL = 20) THEN RESUME 500
            !
            !
            !
500     PRINT 'Error occurred on line';ERL
```

20.0 ERN$

Function

The ERN$ function returns the name of the main program, subprogram, or (*VAX–11 BASIC* only) DEF that was executing when the last error occurred.

Format

```
str-vbl = ERN$
```

Syntax Rules

None.

General Rules

1. In *BASIC–PLUS–2*, if the ERN$ function executes before an error occurs, ERN$ is undefined. When an error occurs, ERN$ is set to the name of the module that caused the error.
2. On *VAX–11 systems*, if the ERN$ function executes before an error occurs or after BASIC executes a RESUME statement, ERN$ returns a null string.

Examples

```
2000      PRINT 'Error in module';ERN$
```

21.0 ERR

Function

The ERR function returns the number of the latest run-time error.

Format

```
int-vbl = ERR
```

Syntax Rules

None.

General Rules

1. If the ERR function is used before an error occurs or after BASIC executes a RESUME statement, results are undefined.
2. The *int-vbl* returned by the ERR function is always a WORD integer in *BASIC–PLUS–2* and a LONG integer in *VAX–11 BASIC*.
3. Appendix B in *BASIC on VAX/VMS Systems, BASIC on RSX–11M/M–PLUS Systems,* or *BASIC on RSTS/E Systems* lists run-time errors and their numbers.

Examples

```
2000      IF (ERR = 11) THEN RESUME 1000
```

22.0 ERT$

Function

The ERT$ function returns explanatory text associated with an error number.

Format

```
str-vbl = ERT$(int-exp)
```

Syntax Rules

None.

General Rules

1. *Int-exp* is an error number. It must be between 0 and 255, inclusive.
2. The ERT$ function can be used at any time to return the text associated with a specified error number.
3. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
2020      PRINT 'Error';ERR;' on line';ERL
          PRINT ERT$(ERR)
```

23.0 EXP

Function

The EXP function returns the value of the mathematical constant "e", raised to a specified power.

Format

```
real-vbl = EXP(real-exp)
```

Syntax Rules

None.

General Rules

1. The EXP function returns the value of "e" raised to the power of *real-exp*.
2. When the default size is SINGLE or DOUBLE, EXP allows arguments between –88 and 88, inclusive. In *VAX–11 BASIC*, if the default size is GFLOAT, EXP allows arguments in the range –709 to 709, inclusive; if the default size is HFLOAT, the arguments can be in the range –11356 to 11355. When the argument exceeds the upper limit of a range, BASIC signals an error. When the argument exceeds the lower limit of a range, the EXP function returns zero and BASIC does not signal an error.
3. BASIC expects the argument of the EXP function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

```
100     A = EXP(4.6)
```

24.0 FIX

Function

The FIX function truncates a floating-point value at the decimal point and returns the integer portion represented as a floating-point value.

Format

```
real-vbl = FIX(real-exp)
```

Syntax Rules

None.

General Rules

1. The FIX function returns the integer portion of a floating-point value, not an integer value.
2. BASIC expects the argument of the FIX function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.
3. If *real-exp* is negative, FIX returns the negative integer portion. For example, FIX(–5.2) returns –5.

Examples

```
200     FIX_VALUE = FIX(-3.333)
210     PRINT FIX(24.566), FIX_VALUE
```

25.0 FORMAT$

Function

The FORMAT$ function converts an expression to a formatted string.

Format

```
str-vbl = FORMAT$(exp, str-exp)
```

Syntax Rules

None.

General Rules

1. The rules for building a format string are the same as those for printing numbers with the PRINT USING statement.

Examples

```
500     PRINT FORMAT$(12345, "**,***")
```

26.0 FSP$

Function

The FSP$ function returns a string describing an open file on a specified channel.

Format

```
str-vbl = FSP$(chnl-exp)
```

Syntax Rules

1. A file must be open on *chnl-exp*. You cannot include a pound sign (#) in *chnl-exp*.
2. The FSP$ function must come immediately after the OPEN statement for the file.

General Rules

1. In *BASIC–PLUS–2*, byte 1 returns the RMS record format field (RFM). In *VAX–11 BASIC*, byte 1 is undefined.
2. In *BASIC–PLUS–2*, bytes 9 and 10 in the returned string contain the RMS Bucketsize (BKS) or RMS Blocksize (BLS) for magnetic tape. Byte 12 is the number of indexes (keys) in the file. In *VAX–11 BASIC*, the FSP$ function returns zeros in bytes 9 through 12.
3. Use the FSP$ function with files opened as ORGANIZATION UNDEFINED. Then use multiple MAP statements to interpret the returned data.
4. See the *BASIC User's Guide* and the *RMS User's Guide* for more information on FSP$ values.

Note

VAX–11 BASIC supports the FSP$ function for compatibility with *BASIC–PLUS–2*. However, you can access the information in bytes 9 through 12 in the returned string more efficiently in *VAX–11 BASIC* by using the USEROPEN clause in the OPEN statement.

Examples

```
500     A$ = FSP$(1)
```

27.0 FSS$ (BASIC–PLUS–2 Only)

Function

The FSS$ function scans a file name string beginning at a specified position and returns a 30–character string describing the file name and status. Because file specifications differ from system to system, the returned string contains system-specific information. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on the values returned by the FSS$ function.

Format

```
str-vbl = FSS$(str-vbl, int-vbl)
```

Syntax Rules

1. *Str-vbl* names the file name string to be scanned.
2. *Int-vbl* specifies the character position at which scanning starts.

General Rules

1. If you specify a floating-point variable for *int-vbl*, BASIC truncates it to an integer of the default size.
2. *Str-vbl* is a 30–character string. See *BASIC on RSX–11M/M–PLUS Systems* and *BASIC on RSTS/E Systems* for information on the encoding of *str-vbl*.

Note

VAX–11 BASIC does not support the FSS$ function. However, the DEFAULTNAME clause in the OPEN statement supplies default file specification components.

Examples

```
100     Y$ = FSS$(A$,B%)
```

28.0 GETRFA

Function

The GETRFA function returns the Record File Address (RFA) of the last record accessed in an RMS file open on a specified channel.

Format

```
rfa-vbl = GETRFA(chnl-exp)
```

Syntax Rules

1. *Rfa-vbl* is a variable of the RFA data type.
2. *Chnl-exp* is the channel number of an open RMS file. You cannot include a pound sign (#) in the channel expression.
3. You must access a record in the file with a GET, FIND, or PUT statement before using the GETRFA function, or BASIC signals "No current record" (ERR=131).

General Rules

1. There must be a file open on the specified *chnl-exp* or BASIC signals an error.
2. You can use the GETRFA function with RMS sequential, relative, indexed, and (except on *RSTS/E systems*) block I/O files.
3. The RFA value returned by the GETRFA function can be used only for assignments to and comparisons with other variables of the RFA data type. Comparisons are limited to equal to (=) and not equal to (<>) relational operations.
4. RFA values cannot be printed or used for any arithmetic operations.

Examples

```
100     DECLARE RFA R_ARRAY(99)
            !
            !
            !
500     FOR I% = 1% TO 100%
                PUT #1
                R_ARRAY(I%) = GETRFA(1)
        NEXT I%
```

29.0 INSTR

Function

The INSTR function searches for a substring within a string. It returns the position of the substring's starting character.

Format

```
int-vbl = INSTR(int-exp, str-exp1, str-exp2)
```

Syntax Rules

None.

General Rules

1. The INSTR function searches *str-exp1*, the main string, for the first occurrence of a substring, *str-exp2*, and returns the position of the substring's first character.
2. *Int-exp* specifies the character position in the main string at which BASIC starts the search.
3. INSTR returns the character position in the main string at which BASIC finds the substring, except in the following situations:
   - If only the substring is null, and if *int-exp* is less than or equal to zero, INSTR returns a value of one.
   - If only the substring is null, and if *int-exp* is equal to or greater than one and less than or equal to the length of the main string, INSTR returns the value of *int-exp*.
   - If only the substring is null, and if *int-exp* is greater than the length of the main string, INSTR returns the main string's length plus one.
   - If the substring is not null, and if *int-exp* is greater than the length of the main string, INSTR returns zero.
   - If only the main string is null, INSTR returns zero.
   - If both the main string and the substring are null, INSTR returns one.
4. If BASIC cannot find the substring, INSTR returns zero.
5. If *int-exp* does not equal one, BASIC still counts from the beginning of the main string to calculate the starting position of the substring. That is, BASIC counts character positions starting at position one, regardless of where you specify the start of the search. For example, if you specify 10 as the start of the search and BASIC finds the substring at position 15, INSTR returns the value 15.
6. If *int-exp* is less than one, BASIC assumes a starting position of one.
7. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Note

VAX–11 BASIC supplies the INSTR function only for compatibility with *BASIC–PLUS–2* and *BASIC–PLUS*. DIGITAL recommends that you use the POS function for substring searches.

Examples

```
300      Y = INSTR(1, ALPHA$, "JKLMN")
```

30.0 INT

Function

The INT function returns the floating-point value of the largest whole number less than or equal to a specified expression.

Format

```
real-vbl = INT(real-exp)
```

Syntax Rules

None.

General Rules

1. If *real-exp* is negative, BASIC returns the largest whole number less than or equal to *real-exp*. For example, INT(–5.3) is –6.
2. BASIC expects the argument of the INT function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.
3. This example contrasts the INT and FIX functions:

```
10   TEST_NUM = -32.7
20   PRINT "INT OF -32.7 IS: "; INT(TEST_NUM)
30   PRINT "FIX OF -32.7 IS: "; FIX(TEST_NUM)
40   END

RUNNH

INT OF -32.7 IS: -33
FIX OF -32.7 IS: -32
```

Examples

```
650     RESULT = INT(6.667)
```

31.0 INTEGER

Function

The INTEGER function converts a numeric expression or numeric string to a specified or default INTEGER data type.

Format

```
int-vbl = INTEGER(exp [, LONG ] )
                      [, BYTE ]
                      [, WORD ]
```

Syntax Rules

1. *Exp* can be either numeric or string. A string expression can contain the ASCII digits 0 through 9, a plus sign (+), or a minus sign (–).

General Rules

1. BASIC evaluates *exp*, then converts it to the specified INTEGER size. If you do not specify a size, BASIC uses the default INTEGER size.
2. If *exp* is a string, BASIC ignores leading and trailing spaces and tabs.
3. The INTEGER function returns a zero when a string argument contains only spaces and tabs, or when it is null.

Examples

```
100     INPUT "Enter a floating-point number";F_P
        PRINT INTEGER(F_P, WORD)
```

32.0 LEFT$

Function

The LEFT$ function extracts a specified substring from a string's left side, leaving the main string unchanged.

Format

```
                 { LEFT  }
str-vbl  =       { LEFT$ }  (str-exp, int-exp)
```

Syntax Rules

None.

General Rules

1. The LEFT$ function extracts a substring from the left of the specified *str-exp* and stores it in *str-vbl*.
2. *Int-exp* specifies the number of characters to be extracted from the left side of the *str-exp*.
3. If *int-exp* is less than one, LEFT$ returns a null string.
4. If *int-exp* is greater than the length of *str-exp*, LEFT$ returns the entire string.
5. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Note

VAX–11 BASIC supplies the LEFT$ function only for compatibility with *BASIC–PLUS* and *BASIC–PLUS–2*. DIGITAL recommends that you use the SEG$ function for substring extraction.

Examples

```
410     SUB_STRING$ = LEFT$(ALPHA$, 5%)
```

33.0 LEN

Function

The LEN function returns an integer value equal to the number of characters in a specified string.

Format

```
int-vbl = LEN(str-exp)
```

Syntax Rules

None.

General Rules

1. If *str-exp* is null, LEN returns a value of zero.
2. The length of *str-exp* includes leading, trailing, and embedded blanks. Tabs in *str-exp* are treated as a single space.
3. The value returned by the LEN function is an integer of the default size.

Examples

```
200     LENGTH = LEN(ALPHA$)
```

34.0 LOC (VAX–11 BASIC Only)

Function

The LOC function returns a longword integer specifying the virtual address of a simple or subscripted variable. For dynamic strings, the LOC function returns the address of the descriptor rather than the address of the data.

Format

```
int-vbl = LOC(vbl)
```

Syntax Rules

1. *Vbl* can be any local or external, simple or subscripted variable.
2. *Vbl* cannot be a virtual array element.

General Rules

1. The LOC function always returns a LONG value.

Examples

```
100     DECLARE LONG B, A
200     A = LOC(B)
```

35.0 LOG

Function

The LOG function returns the natural logarithm (base "e") of a specified number. The LOG function is the inverse of the EXP function.

Format

```
real-vbl = LOG(real-exp)
```

Syntax Rules

None.

General Rules

1. *Real-exp* must be greater than zero. An attempt to find the logarithm of zero or a negative number causes BASIC to signal "Illegal argument in LOG" (ERR=53).
2. The LOG function uses the mathematical constant "e" as a base. BASIC approximates "e" to be 2.718281828459045 (double precision).
3. The LOG function returns the exponent to which "e" must be raised to equal *real-exp*.
4. BASIC expects the argument of the LOG function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size.

Examples

```
10      EXPONENT = LOG(100.35)
```

36.0 LOG10

Function

The LOG10 function returns the common logarithm (base 10) of a specified number.

Format

```
real-vbl = LOG10(real-exp)
```

Syntax Rules

None.

General Rules

1. *Real-exp* must be larger than zero. An attempt to find the logarithm of zero or a negative number causes BASIC to signal "Illegal argument in LOG" (ERR=53).
2. The LOG10 function returns the exponent to which 10 must be raised to equal *real-exp*.
3. BASIC expects the argument of the LOG10 function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

```
600     EXP_BASE_10 = LOG10(250)
```

37.0 MAG

Function

The MAG function returns a number that equals the absolute value of a specified expression. The returned value has the same data type as the expression.

Format

```
vbl = MAG(exp)
```

Syntax Rules

None.

General Rules

1. The returned value is always greater than or equal to zero. The absolute value of zero is zero. The absolute value of a positive number equals that number. The absolute value of a negative number equals that number multiplied by minus one.
2. The MAG function is similar to the ABS function in that it returns the absolute value of a number. The ABS function, however, takes a floating-point argument and returns a floating-point value. The MAG function takes an argument of any numeric data type and returns a value of the same data type as the argument.

Examples

```
100     DECLARE LONG A
200     PRINT MAG(A)
```

38.0 MAGTAPE

Function

The MAGTAPE function permits your program to control unformatted magnetic tape files.

Format

```
int-vbl1 = MAGTAPE(int-const, int-vbl2, chnl-exp)
```

Syntax Rules

1. *Int-const* is an integer between 1 and 9, inclusive, that specifies the code for the MAGTAPE function you want to perform. Function codes are described in Table 23. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on magnetic tape function codes.
2. *VAX–11 BASIC* supports only function code 3, rewind tape. Table 24 explains how to perform other MAGTAPE functions on VAX/VMS Systems.
3. *Int-vbl2* is an integer parameter for function codes 4, 5, and 6.
   - *Int-vbl2* for function 4 is a value from 1 to 32767, inclusive, that specifies the number of records to skip.
   - *Int-vbl2* for function 5 is a value from 1 to 32767, inclusive, that specifies the number of records to backspace.
   - *Int-vbl2* for function 6 specifies the density and/or parity of the magnetic tape drive. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on setting the density and parity of the magnetic tape drive.
4. The *chnl-exp* associated with the magnetic tape must be open.

Table 23: MAGTAPE Function Codes

Code	Meaning
1	Rewind and take tape off-line
2	Write end-of-file (EOF) mark
3	Rewind tape
4	Advance tape a specified number of records
5	Backspace tape a specified number of records
6	Set density and parity
7	Return status of tape
8	Return characteristics of file open on tape (*RSTS/E only*)
9	Rewind when file is closed (*RSTS/E only*)

General Rules

1. You cannot use the MAGTAPE function with RMS files.
2. Function codes 8 and 9 are valid only on *RSTS/E systems*.
3. If *int-const* equals 1, 2, 3, 6, or 9, *int-vbl1* always equals zero.
4. If *int-const* equals 4, *int-vbl1* is an integer of the default size that equals the number of records not skipped.
5. If *int-const* equals 5, *int-vbl1* is an integer of the default size that equals the number of records not backspaced.
6. If *int-const* equals 7, *int-vbl1* is a 16–bit integer that reflects the status of the specified magnetic tape. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on bit values and meaning.
7. If *int-const* equals 9, *int-vbl1* is a 16–bit integer that describes the file characteristics of the specified magnetic tape. See the *RSTS/E Programming Manual* for information on bit values and meaning.
8. On *RSTS/E systems*, the "rewind when file is closed" function (9) must appear after the OPEN statement and before the CLOSE statement associated with the specified magnetic tape. (RSTS)

Table 24: Performing MAGTAPE Functions in VAX–11 BASIC

MAGTAPE Function	VAX–11 BASIC Actions
Write EOF	Close channel
Skip records	Perform GETs, ignore data until reaching desired record
Backspace	Rewind tape, perform GETs, ignore data until reaching desired record
Set density	Use the DCL MOUNT command qualifiers (/DENSITY and /FOREIGN), or the $MOUNT system service
Status	Use the USEROPEN clause in the OPEN statement to access the RAB$L_STS and the RAB$L_STV

Examples

```
200     I = MAGTAPE (1,0,2)
```

39.0 MAR (VAX–11 BASIC Only)

Function

The MAR function returns the current margin width of a specified channel.

Format

```
                 { MAR  }
int-vbl  =       { MAR% }  (chnl-exp)
```

Syntax Rules

None.

General Rules

1. The file associated with *chnl-exp* must be open. You cannot include a pound sign (#) in *chnl-exp*.
2. If *chnl-exp* specifies a terminal, the MAR function returns zero if you have not set a margin width with the MARGIN statement. If you have set a margin width, the MAR function returns that number.
3. The value returned by the MAR function is an integer of the default size.

Examples

```
200     WIDTH = MAR(0)
```

40.0 MID$

Function

The MID$ function extracts a specified substring from the middle of a string, leaving the main string unchanged.

Format

```
                 ⎧MID  ⎫
str-vbl  =       ⎨     ⎬  (str-exp, int-exp1, int-exp2)
                 ⎩MID$ ⎭
```

Syntax Rules

None.

General Rules

1. The MID$ function extracts a substring from *str-exp* and stores it in *str-vbl*. *Int-exp1* specifies the position of the substring's first character. *Int-exp2* specifies the length of the substring.
2. If *int-exp1* is less than one, BASIC assumes a starting position of one.
3. If *int-exp1* is greater than the length of *str-exp*, MID$ returns a null string.
4. If *int-exp2* is greater than the length of *str-exp*, BASIC returns the string that begins at *int-exp1* and includes all characters remaining in the string.
5. If *int-exp2* is less than or equal to zero, MID$ returns a null string.
6. If you specify a floating-point expression for *int-exp1* or *int-exp2*, BASIC truncates it to an integer of the default size.

Note

VAX–11 BASIC supplies the MID$ function only for compatibility with *BASIC–PLUS* and *BASIC–PLUS–2*. DIGITAL recommends that you use the SEG$ function for substring extraction.

Examples

```
220     NEW_STRING$ = MID$(OLD_STRING$, 5, 8)
```

41.0 NOECHO

Function

The NOECHO function disables echoing of input on a terminal.

Format

```
int-vbl = NOECHO(chnl-exp)
```

Syntax Rules

None.

General Rules

1. *Chnl-exp* must specify a terminal. You cannot include a pound sign (#) in *chnl-exp*.
2. If you specify NOECHO, BASIC still accepts characters typed on the terminal as input, but the characters do not echo on the terminal.
3. The NOECHO function is the complement of the ECHO function; that is, NOECHO disables the effect of ECHO and vice versa.
4. NOECHO always returns zero.

Examples

```
500     Y = NOECHO(0)
```

42.0 NUM

Function

The NUM function returns the row number of the last data element transferred into an array by a MAT I/O statement.

Format

```
int-vbl = NUM
```

Syntax Rules

None.

General Rules

1. NUM returns zero if it is invoked before BASIC has executed any MAT I/O statements.
2. For a two-dimensional array, NUM returns an integer specifying the row number of the last data element transferred into the array. For a one-dimensional array, NUM returns the number of elements entered.
3. The value returned by the NUM function is an integer of the default size.

Examples

```
10      ROW_COUNT = NUM
```

43.0 NUM2

Function

The NUM2 function returns the column number of the last data element transferred into an array by a MAT I/O statement.

Format

```
int-vbl = NUM2
```

Syntax Rules

None.

General Rules

1. NUM2 returns zero if it is invoked before BASIC has executed any MAT I/O statements or if the last array element transferred was in a one-dimensional list.
2. The NUM2 function returns an integer specifying the column number of the last data element transferred into an array.
3. The value returned by the NUM2 function is an integer of the default size.

Examples

```
100     COLUMN_COUNT = NUM2
```

44.0 NUM$

Function

The NUM$ function evaluates a numeric expression and returns a string of characters in PRINT statement format, with leading and trailing spaces.

Format

```
str-vbl = NUM$(num-exp)
```

Syntax Rules

None.

General Rules

1. If *num-exp* is positive, the first character in the string expression is a space. If *num-exp* is negative, the first character is a minus sign.
2. The NUM$ function does not include trailing zeros in the returned string. If all digits to the right of the decimal point are zeros, NUM$ omits the decimal point as well.
3. When *num-exp* has an integer portion of six digits or less (for example, 1234.567), BASIC rounds the number to six digits (1234.57). If *num-exp* has seven decimal digits or more, BASIC rounds the number to six digits and prints it in E format.
4. When *num-exp* is between 0.1 and 1, BASIC rounds it to six digits. When *num-exp* is smaller than 0.1, BASIC rounds it to six digits and prints it in E format.
5. If *num-exp* is a longword integer, the returned string can have up to 10 digits.
6. The last character in the returned string is a space.

Examples

```
660     NUMBER$ = NUM$(34.55000/32.4)
```

45.0 NUM1$

Function

The NUM1$ function changes a numeric expression to a numeric character string without leading and trailing spaces.

Format

```
str-vbl = NUM1$(num-exp)
```

Syntax Rules

None.

General Rules

1. The NUM1$ function returns a string consisting of numeric characters and a decimal point that corresponds to the value of *num-exp*. Leading and trailing spaces are not included in the returned string.
2. The NUM1$ function returns:
   - Three digits for BYTE integers
   - Five digits for SINGLE floating-point numbers and WORD integers
   - Ten digits for LONG integers
   - Sixteen digits for DOUBLE floating-point numbers
   - Fifteen digits for GFLOAT floating-point numbers (*VAX–11 BASIC* only)
   - Thirty-three digits for HFLOAT floating-point numbers (*VAX–11 BASIC* only)
3. The NUM1$ function does not produce E notation.

Examples

```
750     NUMBER$ = NUM1$(PI/2)
```

46.0 ONECHR (BASIC–PLUS–2 Only)

Function

The ONECHR function allows single-character input (ODT submode) on a specified channel. This function must be used in conjunction with the GET statement.

Format

```
int-vbl = ONECHR(chnl-exp)
```

Syntax Rules

1. *Chnl-exp* must refer to an open terminal. You cannot include a pound sign (#) in *chnl-exp*.
2. The ONECHR function must be used immediately before the GET statement.

General Rules

1. BASIC disables the ONECHR function immediately after a GET statement executes. Therefore, your program must invoke the ONCHR function for each single character input you want to perform.
2. Control passes to the program as soon as you enter a character. You do not have to type a line terminator.

Examples

```
100     OPEN "TI:" FOR INPUT AS FILE #1%
110     Y% = ONECHR(1%)
120     GET #1%
        MOVE FROM #1%, A$ = 1%
        PRINT A$
```

Note

VAX–11 BASIC does not support the ONECHR function. To perform this function in *VAX–11 BASIC*, you must use the system service SYS$QIO.

47.0 PLACE$

Function

The PLACE$ function explicitly changes the precision of a numeric string. PLACE$ returns a numeric string, truncated or rounded, according to the value of an integer argument you supply.

Format

```
str-vbl = PLACE$(str-exp, int-exp)
```

Syntax Rules

1. *Str-exp* specifies the numeric string you want to process. It can contain up to 60 ASCII digits and an optional decimal point and leading sign.
2. If *str-exp* consists of more than 60 characters, BASIC signals the error "Illegal number" (ERR=52).
3. *Int-exp* specifies the numeric precision of *str-exp*. Table 25 shows examples of rounding and truncation and the values of *int-exp* that produce them.

General Rules

1. *Str-exp* is rounded and/or truncated according to the value of *int-exp*.
2. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
   - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
   - If *int-exp* is zero, BASIC rounds to the nearest unit.
   - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is –1, for example, BASIC moves the decimal point one place to the left, then rounds to units. If *int-exp* is –2, rounding occurs two places to the left of the decimal point; BASIC moves the decimal point two places to the left, then rounds to tens.
3. If *int-exp* is between 9940 and 10060, truncation occurs:
   - If *int-exp* is 10000, BASIC truncates the number at the decimal point.
   - If *int-exp* is greater than 10000 (10000 plus n) BASIC truncates the numeric string n places to the right of the decimal point. For example, if *int-exp* is 10001 (10000 plus 1), BASIC truncates the number starting one place to the right of the decimal point. If *int-exp* is 10002 (10000 plus 2), BASIC truncates the number starting two places to the right of the decimal point, and so on.

- If *int-exp* is less than 10000 (10000 minus n), BASIC truncates the numeric string n places to the left of the decimal point. For example, if *int-exp* is 9999 (10000 minus 1), BASIC truncates the number starting one place to the left of the decimal point. If *int-exp* is 9998 (10000 minus 2), BASIC truncates the number starting two places to the left of the decimal point, and so on.

4. If *int-exp* is not between –60 and 60 or 9940 and 10060, BASIC returns zero.

5. If you specify a floating-point expresstion for *int-exp*, BASIC truncates it to an integer of the default size.

6. Table 25 shows examples of rounding and truncation and the values of *int-exp* that produce them. The number used is 123456.54321.

Examples

```
500     NUMBER$ = PLACE$(OLD_NUMBER$, 10001)
```

Table 25: Rounding and Truncation of 123456.654321

Int-exp	Effect	Value Returned
–5	Rounded to 100,000s and truncated	1
–4	Rounded to 10,000s and truncated	12
–3	Rounded to 1000s and truncated	123
–2	Rounded to 100s and truncated	1235
–1	Rounded to 10s and truncated	12346
0	Rounded to units and truncated	123457
1	Rounded to tenths and truncated	123456.7
2	Rounded to hundredths and truncated	123456.65
3	Rounded to thousandths and truncated	123456.654
4	Rounded to ten-thousandths and truncated	123456.6543
5	Rounded to hundred-thousandths and truncated	123456.65432
9,995	Truncated to 100,000s	1
9,996	Truncated to 10,000s	12
9,997	Truncated to 1000s	123
9,998	Truncated to 100s	1234
9,999	Truncated to 10s	12345
10,000	Truncated to units	123456
10,001	Truncated to tenths	12345.6
10,002	Truncated to hundredths	123456.65
10,003	Truncated to thousandths	123456.654
10,004	Truncated to ten-thousandths	123456.6543
10,005	Truncated to hundred-thousandths	123456.65432

48.0 POS

Function

The POS function searches for a substring within a string and returns the substring's starting character position.

Format

```
int-vbl = POS(str-exp1, str-exp2, int-exp)
```

Syntax Rules

None.

General Rules

1. The POS function searches *str-exp1*, the main string, for the first occurrence of *str-exp2*, the substring, and returns the position of the substring's first character.
2. *Int-exp* specifies the character position in the main string at which BASIC starts the search.
3. If *int-exp* is greater than the length of the main string, POS returns zero.
4. POS always returns the character position in the main string at which BASIC finds the substring:
   - If only the substring is null, and if *int-exp* is less than or equal to zero, POS returns a value of one.
   - If only the substring is null, and if *int-exp* is equal to or greater than one and less than or equal to the length of the main string, POS returns the value of *int-exp*.
   - If only the substring is null and if *int-exp* is greater than the length of the main string, POS returns the main string's length plus one.
   - If only the main string is null, POS returns zero.
   - If both the main string and the substring are null, POS returns one.
5. If BASIC cannot find the substring, POS returns zero.
6. If *int-exp* is less than one, BASIC assumes a starting position of one.
7. If *int-exp* does not equal one, BASIC still counts from the string's beginning to calculate the starting position of the substring. That is, BASIC counts character positions starting at position one, regardless of where you specify the start of the search. For example, if you specify 10 as the start of the search and BASIC finds the substring at position 15, POS returns the value 15.

8. If you know that the substring is not near the beginning of the string, specifying a starting position greater than one speeds program execution by reducing the number of characters BASIC must search.

9. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
400      Y = POS(ALPHA$, "JKLMN", 1)
```

49.0 PROD$

Function

The PROD$ function returns a numeric string that is the product of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

Format

```
str-vbl = PROD$(str-exp1, str-exp2, int-exp)
```

Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to process. They can contain up to 60 ASCII digits and an optional decimal point and leading sign.
2. If *str-exp* consists of more than 60 characters, BASIC signals the error "Illegal number" (ERR=52).
3. *Int-exp* specifies the numeric precision of *str-exp*. Table 25 shows examples of rounding and truncation and the values of *int-exp* that produce them.

General Rules

1. *Str-exp* is rounded and/or truncated according to the value of *int-exp*.
2. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
   - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
   - If *int-exp* is zero, BASIC rounds to the nearest unit.
   - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is –1, for example, BASIC moves the decimal point one place to the left, then rounds to units. If *int-exp* is –2, rounding occurs two places to the left of the decimal point; BASIC moves the decimal point two places to the left, then rounds to tens.
3. If *int-exp* is between 9940 and 10060, truncation occurs:
   - If *int-exp* is 10000, BASIC truncates the number at the decimal point.
   - If *int-exp* is greater than 10000 (10000 plus n), BASIC truncates the numeric string n places to the right of the decimal point. For example, if *int-exp* is 10001 (10000 plus 1), BASIC truncates the number starting one place to the right of the decimal point. If *int-exp* is 10002 (10000 plus 2), BASIC truncates the number starting two places to the right of the decimal point, and so on.

(continued on next page)

- If *int-exp* is less than 10000 (10000 minus n), BASIC truncates the numeric string n places to the left of the decimal point. For example, if *int-exp* is 9999 (10000 minus 1), BASIC truncates the number starting one place to the left of the decimal point. If *int-exp* is 9998 (10000 minus 2), BASIC truncates the number starting two places to the left of the decimal point, and so on.

4. If *int-exp* is not between –60 and 60 or 9940 and 10060, BASIC returns zero.

5. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

6. Table 25 shows examples of rounding and truncation and the values of *int-exp* that produce them. The number used is 123456.654321.

Examples

```
300     PRODUCT$ = PROD$("88793", Z$, 0)
```

50.0 QUO$

Function

The QUO$ function returns a numeric string that is the quotient of two numeric strings. The precision of the returned numeric string depends on the value of an integer argument.

Format

```
str-vbl = QUO$(str-exp1, str-exp2, int-exp)
```

Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to process. They can contain up to 60 ASCII digits and an optional decimal point and leading sign.
2. If *str-exp* consists of more than 60 characters, BASIC signals the error "Illegal number" (ERR=52).
3. *Int-exp* specifies the numeric precision of *str-exp*. Table 25 shows examples of rounding and truncation and the values of *int-exp* that produce them.

General Rules

1. *Str-exp* is rounded and/or truncated according to the value of *int-exp*.
2. If *int-exp* is between –60 and 60, rounding and truncation occur as follows:
   - For positive integer expressions, rounding occurs to the right of the decimal place. For example, if *int-exp* is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If *int-exp* is 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
   - If *int-exp* is zero, BASIC rounds to the nearest unit.
   - For negative integer expressions, rounding occurs to the left of the decimal point. If *int-exp* is –1, for example, BASIC moves the decimal point one place to the left, then rounds to units. If *int-exp* is –2, rounding occurs two places to the left of the decimal point; BASIC moves the decimal point two places to the left, then rounds to tens.
3. If *int-exp* is between 9940 and 10060, truncation occurs:
   - If *int-exp* is 10000, BASIC truncates the number at the decimal point.
   - If *int-exp* is greater than 10000 (10000 plus n), BASIC truncates the numeric string n places to the right of the decimal point. For example, if *int-exp* is 10001 (10000 plus 1), BASIC truncates the number starting one place to the right of the decimal point. If *int-exp* is 10002 (10000 plus 2), BASIC truncates the number starting two places to the right of the decimal point, and so on.

(continued on next page)

- If *int-exp* is less than 10000 (10000 minus n), BASIC truncates the numeric string n places to the left of the decimal point. For example, if *int-exp* is 9999 (10000 minus 1), BASIC truncates the number starting one place to the left of the decimal point. If *int-exp* is 9998 (10000 minus 2), BASIC truncates the number starting two places to the left of the decimal point, and so on.

4. If *int-exp* is not between –60 and 60 or 9940 and 10060, BASIC returns zero.
5. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.
6. Table 25 shows examples of rounding and truncation and the values of *int-exp* that produce them. The number used is 123456.654321.

Examples

```
200      QUOTIENT$ = QUO$("453.221", "30", 10000)
```

51.0 RAD$

Function

The RAD$ function converts a specified integer to a 3–character string in Radix–50 format.

Format

```
str-vbl = RAD$(int-vbl)
```

Syntax Rules

None.

General Rules

1. The RAD$ function converts *int-vbl* to a 3–character string in Radix–50 format and stores it in *str-vbl*. Radix–50 format allows you to store three characters of data as a 2–byte integer.
2. See Appendix C in *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on the Radix–50 character set and ASCII/Radix–50 equivalents.
3. *VAX–11 BASIC* supports the RAD$ function, but not its complement, the FSS$ function. DIGITAL recommends that you use Run-Time Library routines for Radix–50 operations. VAX
4. If you specify a floating-point variable for *int-vbl*, BASIC truncates it to an integer of the default size.

Examples

```
100     RADIX$ = RAD$(999)
```

52.0 RCTRLC

Function

The RCTRLC function disables CTRL/C trapping.

Format

```
int-vbl = RCTRLC
```

Syntax Rules

None.

General Rules

1. After BASIC executes the RCTRLC function, a CTRL/C typed at the terminal returns you to command level (BASIC or monitor).
2. RCTRLC always returns a zero.

Examples

```
200     Y = RCTRLC
```

53.0 RCTRLO

Function

The RCTRLO function cancels the effect of a CTRL/O typed on a specified channel.

Format

```
int-vbl = RCTRLO (chnl-exp)
```

Syntax Rules

None.

General Rules

1. *Chnl-exp* must refer to a terminal.
2. RCTRLO has no effect if the specified channel is open to a device that does not use the CTRL/O convention.
3. If you type a CTRL/O to cancel terminal output, nothing is printed on the specified terminal until your program executes the RCTRLO or until you type another CTRL/O, at which time normal terminal output resumes.
4. The RCTRLO function always returns a zero.

Examples

```
10      PRINT "A" FOR I% = 1% TO 100%
        Y% = RCTRLO(0%)
        PRINT "Normal output is resumed"
```

55.0 RECOUNT

Function

The RECOUNT function returns the number of characters transferred by the last input operation.

Format

```
int-vbl = RECOUNT
```

Syntax Rules

None.

General Rules

1. The RECOUNT value is set by every input operation on any channel, including channel zero.
   - After an input operation from your terminal, RECOUNT contains the number of characters (bytes), including line terminators, transferred.
   - After accessing a file record, RECOUNT contains the number of characters in the record.
2. Because RECOUNT is reset by every input operation on any channel, use the RECOUNT function to copy the RECOUNT value to a different storage location before executing another input operation.
3. If an error occurs during an input operation, the value of RECOUNT is undefined.
4. RECOUNT is unreliable after a CTRL/C interrupt because the CTRL/C trap may have occurred before BASIC set the value for RECOUNT.
5. The RECOUNT function returns a LONG value in *VAX–11 BASIC* and a WORD value in *BASIC–PLUS–2*.

Examples

```
200     CHARACTER_COUNT = RECOUNT
        PRINT CHARACTER_COUNT;' characters received'
```

56.0 RIGHT$

Function

The RIGHT$ function extracts a substring from a string's right side, leaving the main string unchanged.

Format

```
                 { RIGHT  }
str-vbl  =       { RIGHT$ }  (str-exp, int-exp)
```

Syntax Rules

None.

General Rules

1. The RIGHT$ function extracts a substring from *str-exp* and stores the substring in *str-vbl*. The substring begins with the character in the position specified by *int-exp* and ends with the rightmost character in the string.
2. If *int-exp* is less than or equal to zero, RIGHT$ returns the entire string.
3. If *int-exp* is greater than the length of *str-exp*, RIGHT$ returns a null string.
4. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
600     NEW_STRING$ = RIGHT$(ALPHA$, 21)
```

Note

VAX–11 BASIC includes the RIGHT$ function only for compatibility with *BASIC–PLUS* and *BASIC–PLUS–2*. DIGITAL recommends using the SEG$ function for substring extraction.

57.0 RND

Functions

The RND function returns a random number greater than or equal to zero and less than one.

Format

```
real-vbl = RND
```

Syntax Rules

None.

General Rules

1. The RND function returns a pseudorandom number if not preceded by a RANDOMIZE statement; that is, each time a program runs, BASIC generates the same random number or series of random numbers.
2. If the RND function is preceded by a RANDOMIZE statement, BASIC generates a different random number or series of numbers each time a program executes.
3. In *BASIC–PLUS–2*, the RND function returns a floating-point value of the default size. In *VAX–11 BASIC*, RND always returns a single-precision value.

Examples

```
990     R_NUM = RND
```

58.0 SEG$

Function

The SEG$ function extracts a substring from a main string, leaving the original string unchanged.

Format

```
str-vbl = SEG$(str-exp, int-exp1, int-exp2)
```

General Rules

1. BASIC extracts the substring from *str-exp*, the main string, and stores the substring in *str-vbl*. The substring begins with the character in the position specified by *int-exp1* and ends with the character in the position specified by *int-exp2*.
2. If *int-exp1* is less than one, BASIC assumes a value of one.
3. If *int-exp1* is greater than *int-exp2* or the length of *str-exp*, the SEG$ function returns a null string.
4. If *int-exp1* equals *int-exp2*, the SEG$ function returns the character at the position specified by *int-exp1*.
5. Unless *int-exp2* is greater than the length of *str-exp*, the length of the returned substring equals *int-exp2* minus *int-exp1* plus one. If *int-exp2* is greater than the length of *str-exp*, the SEG$ function returns all characters from the position specified by *int-exp1* to the end of *str-exp*.
6. If you specify a floating-point expression for *int-exp1* or *int-exp2*, BASIC truncates it to an integer of the default size.

Examples

```
300     CENTER$ = SEG$(ALPHA$, 15, 20)
```

59.0 SGN

Function

The SGN function determines whether a numeric expression is positive, negative, or zero. It returns a one if the expression is positive, a minus one if the expression is negative, and zero if the expression is zero.

Format

```
int-vbl = SGN(real-exp)
```

Syntax Rules

None.

General Rules

1. If *real-exp* does not equal zero, SGN returns ABS(*real-exp*)/*real-exp*.
2. If *real-exp* equals zero, SGN returns zero.
3. SGN returns an integer of the default size.

Examples

```
750     SIGN = SGN(-4535/6-3000)
```

60.0 SIN

Function

The SIN function returns the sine, in radians, of an angle.

Format

```
real-vbl = SIN(real-exp)
```

Syntax Rules

None.

General Rules

1. The returned value is between minus one and one.
2. BASIC expects the argument of the ABS function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

```
100     S1_ANGLE = SIN(PI/2)
```

61.0 SPACE$

Function

The SPACE$ function creates a string containing a specified number of spaces.

Format

```
str-vbl = SPACE$(int-exp)
```

Syntax Rules

None.

General Rules

1. *Int-exp* specifies the number of spaces in the returned string.
2. BASIC treates an *int-exp* less than zero as zero.
3. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
880     FILLER$ = SPACE$(32)
```

62.0 SQR

Function

The SQR function returns the square root of a positive number.

Format

```
real-vbl = { SQRT } (real-exp)
           { SQR  }
```

Syntax Rules

None.

General Rules

1. *VAX–11 BASIC* signals the error "Imaginary square roots" (ERR=54) and program execution stops when *real-exp* is negative.

2. *BASIC–PLUS–2* returns the warning message "%Imaginary square roots" and the square root of the absolute value of the expression when *real-exp* is negative. The program does not stop executing.

3. BASIC assumes that the argument of the SQR function is a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC returns a value of the default floating-point size.

Examples

```
425     ROOT = SQR(35*37)
```

63.0 STATUS

Function

The STATUS function returns an integer value containing information about the last opened channel. Your program can test each bit to determine the status of the channel.

Format

```
int-vbl = STATUS
```

Syntax Rules

None.

General Rules

1. The STATUS function returns a WORD integer in *BASIC–PLUS–2* and a LONG integer in *VAX–11 BASIC*.
2. The value returned by the STATUS function is undefined until BASIC executes an OPEN statement.
3. The STATUS value is set by every input operation on any channel. Therefore, the STATUS value should be copied to a different storage location before your program executes another input operation.
4. The syntax for STATUS is the same for *VAX–11, RSTS/E, and RSX–11M/M–PLUS systems*. However, the returned information is different on every system.
5. Depending on the error, the STATUS function on *RSX–11M/M–PLUS systems* displays a value representing one of the following: **RSX**

   - The RMS–11 primary status field (STS) or the RMS secondary status field (STV). See the *RMS–11 MACRO User's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation (set by the DEV field of the FAB). See the *RMS–11 MACRO User's Guide* for more information.
   - The Directive Status Word ($DSW) and its corresponding error code, in the event of a directive error. See the *RSX–11M/M–PLUS Mini Reference* for the error codes.
   - The STATUS field of a QIO. See the *RSX–11M/M–PLUS I/O Drivers Reference Manual* for more information.
   - The first word of a GETLUN directive describing device characteristics. See the *RSX–11M/M–PLUS Executive Reference Manual*.

   See *BASIC on RSX–11M/M–PLUS Systems* for information on STATUS values set for an OPEN file operation with no errors.

RSTS

6. Depending on the error, the STATUS function on *RSTS/E systems* displays a value representing one of the following:

   - The RMS–11 primary status field (STS) or the RMS secondary status field (STV). See the *RMS–11 MACRO User's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation (set by the DEV field of the FAB). See the *RMS–11 MACRO User's Guide* for more information.

   For OPEN operations where no errors occur, the status word describes the device characteristics of the FIRQB and FQFLAG field. The first 7 bits describe the device, and bits 7 through 15 describe characteristics of the OPEN statement. See the *BASIC–PLUS Language Manual* and the *RSTS/E System Directives Manual* for more information on STATUS values.

VAX

7. In *VAX–11 BASIC*, if an error occurs during an input operation, the value of STATUS is undefined. When no error occurs, the six low-order bits of the returned value contain information about the type of device accessed by the last input operation. Table 26 lists STATUS bits set by *VAX–11 BASIC*.

Table 26: VAX–11 BASIC STATUS Bits

Bit Set	Device Type
0	Record-oriented device
1	Carriage-control device
2	Terminal
3	Directory device
4	Single directory device
5	Sequential block-oriented device (magtape)

Examples

```
150     Y% = STATUS
```

64.0 STR$

Function

The STR$ function changes a numeric expression to a numeric character string without leading and trailing spaces.

Format

```
str-vbl = STR$(num-exp)
```

Syntax Rules

None.

General Rules

1. If *num-exp* is negative, the first character in the returned string is a minus sign.
2. Like the NUM$ function, the STR$ function produces E notation. Unlike the NUM$ function, the STR$ function does not return leading or trailing spaces.
3. Like the NUM1$ function, the STR$ function does not return leading or trailing spaces. Unlike the NUM1$ function, the STR$ function produces E notation.
4. When you print a number whose integer portion is six digits or less (for example, 1234.567), BASIC rounds the number to six digits (1234.57). If a number has seven integer digits or more, BASIC rounds the number to six digits and prints it in E format.
5. When you print a number with magnitude between 0.1 and 1, BASIC rounds it to six digits. When you print a number with magnitude smaller than 0.1, BASIC rounds it to six digits and prints it in E format.

Examples

```
800     Z$ = STR$(65)
```

65.0 STRING$

Function

The STRING$ function creates a string containing a specified number of identical characters.

Format

```
str-vbl = STRING$(int-exp1, int-exp2)
```

Syntax Rules

None.

General Rules

1. *Int-exp1* specifies the character string's length. *VAX–11 BASIC* signals the error "String too long" (ERR=227) if *int-exp1* is greater than 65535. *BASIC–PLUS–2* signals the error "Integer error" (ERR=51) if *int-exp* is greater than 32767.
2. If *int-exp1* is less than or equal to zero, BASIC treats it as zero.
3. *Int-exp2* is the decimal ASCII value of the character that makes up the string. This value is treated modulo 256.
4. BASIC treats all arguments as unsigned 8–bit integers. Negative numbers are treated as the two's complement (for example, –1 is treated as 255).
5. If either *int-exp1* or *int-exp2* is a floating-point expression, BASIC truncates it to an integer of the default size.

Examples

```
340     A_STRING$ = STRING$(10, 65)
```

66.0 SUM$

Function

The SUM$ function returns a string whose value is the sum of two numeric strings.

Format

```
str-vbl = SUM$(str-exp1, str-exp2)
```

Syntax Rules

1. *Str-exp1* and *str-exp2* specify the numeric strings you want to process. They can contain up to 54 ASCII digits and an optional decimal point and leading sign.

General Rules

1. BASIC adds *str-exp2* to *str-exp1* and stores the result in *str-vbl*.
2. If *str-exp1* and *str-exp2* are integers, *str-vbl* takes the precision of the larger string unless trailing zeros generate that precision.
3. If *str-exp1* and *str-exp2* are decimal fractions, *str-vbl* takes the precision of the more precise fraction unless trailing zeros generate that precision.
4. SUM$ omits trailing zeros to the right of the decimal point.
5. The sum of two floating-point numbers takes precision as follows:
   - The sum of the integer parts takes the precision of the larger part.
   - The sum of the decimal fraction part takes the precision of the more precise part.
6. SUM$ truncates leading and trailing zeros.

Examples

```
600     SIGMA$ = SUM$("234.444", A$)
```

67.0 SWAP%

Function

The SWAP% function transposes a WORD integer's bytes.

Format

```
int-vbl = SWAP%(int-exp)
```

Syntax Rules

1. SWAP% is a WORD function. BASIC evaluates *int-exp* and converts it to the WORD data type, if necessary.

General Rules

1. BASIC transposes the bytes of *int-exp* and returns a WORD integer.

Examples

```
500     S_25 = SWAP%(3)
```

68.0 SYS (BASIC–PLUS–2 on RSTS/E Only)

Function

The SYS function lets you perform special I/O functions, establish special characteristics for a job, set terminal characteristics, and cause the monitor to execute special operations.

Format

```
str-vbl = SYS(str-exp)
```

Syntax Rules

1. *Str-exp* is a *RSTS/E* SYS call code. See the *RSTS/E Programming Language* manual for a complete list of SYS call codes and their meanings.

General Rules

1. Because SYS calls request that the RSTS/E monitor perform an operation, often the function performed has no counterpart on other host systems. However, for compatibility with *RSTS/E BASIC–PLUS*, *VAX–11 BASIC* supports a subset of SYS calls. Table 27 lists the *VAX–11 BASIC* subset of *RSTS/E* SYS calls.

Table 27: VAX–11 BASIC Subset of RSTS/E SYS Calls

Code	Function
0	Cancel CTRL/O
1	Not implemented
2	Enable echo
3	Disable echo
4	Not implemented
5	Exit with no prompt
6	Call File Processor
7	Get core common; can be used only between BASIC images
8	Put core common; can be used only between BASIC images
9	Exit and clear program
10	Reserved
11	Cancel type ahead
12	Not implemented
13	Reserved
14	Not implemented

(continued on next page)

Table 27: VAX–11 BASIC Subset of RSTS/E SYS Calls (Cont.)

These FIP calls (and only these) are also supported:

Code	Function
–23	Terminate file name string scan
–13	Set priority (can set only priority; requires ALTPRI privilege)
–10	Begin file name string scan
–7	Enable CTRL/C trap
9	Get error message (*VAX–11 BASIC* error message)
10	Assign a device
11	Deassign a device
12	Deassign all devices
18	Send/receive message (requires SYSNAM privilege)
22	Send/receive message (cannot get job number, privileges, or receive selection; cannot use DECnet; requires PRMMBX privilege)

Examples

```
100     OPEN User_Keyboard$ AS FILE #1
200     Tmp$ = SYS(CHR$(11)) ! Cancel any typeahead from user
300     LINUT 'Enter the first line of text';User_input$
```

69.0 TAB

Function

When used with the PRINT statement, the TAB function moves the cursor or print mechanism right to a specified column.

Format

```
str-vbl = TAB(int-exp)
```

Syntax Rules

1. *Int-exp* specifies the column number of the cursor or print mechanism.

General Rules

1. You cannot TAB beyond the current MARGIN restriction.
2. The leftmost column position is zero.
3. If *int-exp* is less than the current cursor position, the TAB function has no effect.
4. The TAB function can move the cursor or print mechanism only from the left to the right.
5. You can use more than one TAB function in the same PRINT statement.
6. Use semicolons to separate multiple TAB functions in a single statement. If you use commas, BASIC moves to the next print zone before executing the TAB function.
7. The TAB function is valid only for terminals.
8. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
200     PRINT A$;TAB(15);B$;TAB(30);"HELLO"
```

70.0 TAN

Function

The TAN function returns the tangent, in radians, of an angle.

Format

```
real-vbl = TAN(real-exp)
```

Syntax Rules

None.

General Rules

1. BASIC expects the argument of the ABS function to be a *real-exp*. When the argument is a *real-exp*, BASIC returns a value of the same floating-point size. When the argument is not a *real-exp*, BASIC converts the argument to the default floating-point size and returns a value of the default floating-point size.

Examples

```
550      X = TAN(2*PI)
```

71.0 TIME

Function

The TIME function returns the time of day (in seconds) as a floating-point number. On *VAX–11* and *RSTS/E systems* the TIME function can also return CPU time and device connect time.

Format

```
real-vbl = TIME(int-exp)
```

Syntax Rules

None.

General Rules

1. The value returned by the TIME function depends on the value of *int-exp*.

2. If *int-exp* equals 0, TIME returns the number of seconds since midnight.

3. *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems* accepts only an argument of zero. All other arguments to the TIME function are undefined and cause BASIC to signal "Not implemented" (ERR=250). RSX

4. *VAX–11 BASIC* and *BASIC–PLUS–2* on *RSTS/E systems* also accept arguments from 1 through 4 and return values as shown in Table 28. All other arguments to the TIME function are undefined and cause BASIC to signal "Not implemented" (ERR=250).

5. In *BASIC–PLUS–2*, the TIME function returns a floating-point value of the default size. In *VAX–11 BASIC*, TIME always returns a single-precision value.

6. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Table 28: TIME Function Values

Argument Value:	VAX–11 BASIC Returns:	BASIC–PLUS–2 on RSTS/E Systems Returns:
1	The current job's CPU time in tenths of a second	The current job's CPU time in tenths of a second
2	The current job's connect time in minutes	The current job's connect time in minutes
3	Zero	Kilo-core ticks
4	Zero	Device time in minutes

Examples

```
150     PRINT TIME(0)
```

72.0 TIME$

Function

The TIME$ function returns a string displaying the time of the day.

Format

```
str-vbl = TIME$(int-exp)
```

Syntax Rules

None.

General Rules

1. If *int-exp* equals zero, TIME$ returns the current time of day.
2. *Int-exp* is the number of minutes before midnight. *Str-vbl* is the time of day.
3. The value of *int-exp* must be in the range 0 to 1440, inclusive, or BASIC signals an error.
4. In *VAX–11 BASIC* the TIME$ function uses a 12–hour, AM/PM clock. Before 12:00 noon, TIME$ returns HH:MM AM, and after 12:00 noon, HH:MM PM.
5. In *BASIC–PLUS–2* the TIME$ function uses either an AM/PM or a 24–hour clock. The clock type is an installation option.
6. On *RSTS/E systems* only, the clock type can also be set by the system manager at system start-up time.
7. If you specify a floating-point expression for *int-exp*, BASIC truncates it to an integer of the default size.

Examples

```
200     CURRENT_TIME$ = TIME$(0)
```

73.0 TRM$

Function

The TRM$ function removes all trailing blanks and tabs from a specified string.

Format

```
str-vbl = TRM$(str-exp)
```

Syntax Rules

None.

General Rules

1. The returned *str-vbl* is the same as *str-exp* with all the trailing blanks and tabs removed.

Examples

```
600     NEW_STRING$ = TRM$(OLD_STRING$)
```

74.0 VAL

Function

The VAL function converts a numeric string to a floating-point value.

Format

```
real-vbl = VAL(str-exp)
```

Syntax Rules

1. *Str-exp* can contain the ASCII digits 0 through 9, uppercase E, and an optional decimal point and leading sign.
2. BASIC ignores leading, trailing, and embedded spaces and tabs.

General Rules

1. If *str-exp* is null, or contains only spaces and tabs, VAL returns a zero.
2. The value returned by the VAL function is of the default floating-point size.

Examples

```
100     REAL_NUM = VAL("990.32")
```

75.0 VAL%

Function

The VAL% function converts a numeric string to an integer.

Format

```
int-vbl = VAL%(str-exp)
```

Syntax Rules

1. *Str-exp* can contain the ASCII digits 0 through 9 and an optional leading sign.
2. BASIC ignores leading, trailing, and embedded spaces and tabs.

General Rules

1. If *str-exp* is null or contains only spaces and tabs, VAL% returns a value of zero.
2. The value returned by the VAL% function is an integer of the default size.
3. If *str-exp* contains a decimal point, BASIC signals the error "Illegal number" (ERR=52).

Examples

```
100     A = VAL%("999")
```

76.0 XLATE

Function

The XLATE function translates one string to another by referencing a table string you supply.

Format

```
str-vbl = XLATE(str-exp1, str-exp2)
```

Syntax Rules

1. *Str-exp1* is the input string. *Str-exp2* is the table string, and *str-vbl* is the returned string.

General Rules

1. *Str-exp2* can contain up to 256 ASCII characters, numbered from 0 to 255; the position of each character in the string corresponds to an ASCII value. Because zero is a valid ASCII value (null), the first position in the table string is position zero.
2. XLATE scans *str-exp1* character by character, from left to right. It finds the ASCII value n of the first character in *str-exp1* and extracts the character it finds at position n in *str-exp2*. XLATE then appends the character from *str-exp2* to *str-vbl*. XLATE continues this process, character by character, until the end of *str-exp1* is reached.
3. The output string may be smaller than the input string.
   - XLATE does not translate nulls. If the character at position n in *str-exp2* is a null, XLATE does not append that character to *str-vbl*.
   - If the ASCII value of the input character is outside the range of positions in *str-exp2*, XLATE does not append any character to *str-vbl*.

Examples

```
100     OUTPUT$ = XLATE(INPUT$, TABLE$)
```

PART VI
BASIC–PLUS–2
Debugger Commands

Note

This section describes *BASIC–PLUS–2* debugger commands. See *BASIC on VAX/VMS Systems* for information on the *VAX–11 Symbolic Debugger*.

BASIC–PLUS–2 debugger commands help you locate run-time errors and debug program modules interactively in the BASIC environment or from monitor level. To use debugger commands, you must compile or run at least one program module using the /DEBUG qualifier.

When you run a task-built program, execution stops at the first line number of the first module compiled with the /DEBUG qualifier and control passes to the debugger. When you run a program in the BASIC environment, control passes to the debugger when the first line number of the program executed with the RUN/DEBUG command is encountered or when the first line number of an object module compiled with the /DEBUG qualifier and loaded with the LOAD command is encountered.

When control passes to the *BASIC–PLUS–2* debugger, an identifying message and prompt are displayed:

```
DEBUG:module-name

#
```

Module-name is the name of the first program module encountered that was compiled with the /DEBUG qualifier or executed with the RUN/DEBUG command. The pound sign (#) prompt signals you to enter debugger commands. For example:

```
#BREAK 300 RET
#TRACE RET
#CONTINUE RET
```

In the example on the previous page, the BREAK command will cause execution to stop at the first statement on line 300; the TRACE command will cause the line numbers and statement numbers to be displayed as they execute. The CONTINUE command causes the program module to execute until line 300; input, output, and the processing proceeds as usual until the breakpoint is reached. When the BREAK command has successfully executed, the debugger displays a message identifying your current position in the program module and prompts for another debugger command. For example:

```
at line 100 statement 1
at line 100 statement 2
at line 100 statement 3
at line 200 statement 1
at line 200 statement 2
BREAK at line 300 statement 1 DEBTST

#
```

The identifying message names the debugger command that stopped program execution (BREAK), the line number and statement where execution stopped, and the name of the currently executing module (DEBTST in the above example). If the main program is executing, no module name is displayed. The # prompt signals you to enter more debugger commands.

Use the EXIT command to exit from the debugger and end program execution.

When you compile a program with the /DEBUG qualifier, BASIC links the debugger program module from the *BASIC–PLUS–2* OTS to your program. This increases the size of your task by at least 4K bytes. When you task-build the program, the debugger records are included in the executable task image. When you run the executable image, *BASIC–PLUS–2* accesses these records and you can use the debugger commands described in the following sections.

No debugger records are generated for program modules not compiled with the /DEBUG qualifier. Thus, you cannot access information, trace module execution, or establish breakpoints in modules not compiled with the /DEBUG qualifier. You can, however, use debugger commands to access information about the entire task if you compile at least one program module with the /DEBUG qualifier.

After you have debugged your module and changed the source code where necessary, compile the module without the /DEBUG qualifier to reduce memory requirements.

Debugger commands are described on the following pages. All debugger commands except BREAK ON can be abbreviated to three letters.

For an example of a complete debugging session and more information on using the *BASIC–PLUS–2* debugger, see *BASIC on RSTS/E Systems* or BASIC on RSX–11M/M–PLUS Systems.

1.0 BREAK (BASIC–PLUS–2)

Function

The BREAK command lets you stop program execution at program line numbers, particular statements, or at the beginning of CALL statements, user-defined functions; and FOR, UNTIL, and WHILE loops. The program stops before executing the specified breakpoint.

Format

```
BREAK { ON block                    }
      { [ ON ] stmnt-break,...      }

        { CALL }
block:  { DEF  }
        { LOOP }

stmnt-break: lin-num [ .stmnt-num ] [ ;mod-nam ]
```

Syntax Rules

1. The BREAK command with no parameters sets a breakpoint at each line number. The program stops at each line number before executing any statements on the line.

2. *Block* specifies a block statement. The ON keyword is required. You can specify only one block statement in a BREAK ON statement:

   - BREAK ON CALL stops execution each time BASIC executes a CALL statement to a subprogram. The program stops before any statements in the subprogram execute. If you are executing a task-built program, both the calling and the called program must be compiled with the /DEBUG qualifier or the BREAK ON CALL command has no effect. If you are executing a program in the BASIC environment, the called program must be compiled with the /DEBUG qualifier.

   - BREAK ON DEF stops execution each time BASIC encounters a user-defined function in a module compiled with the /DEBUG qualifier. The statement stops before any statements in the function execute.

   - BREAK ON LOOP stops execution each time BASIC encounters a FOR, WHILE, or UNTIL statement or modifier. The program stops each time the program loops back to the loop statement. The program stops after the loop is initialized or incremented, but before any statements in the loop execute.

3. *Stmnt-break* specifies a particular line number or statement where execution is to stop.

   - *Lin-num* specifies a program line.
   - *Stat-num* specifies a particular statement associated with *lin-num*. The period (.) is required and must immediately follow the line number. BASIC signals an error if you include a space between *lin-num* and *stat-num*. The cross-reference listing file lists statements on multi-statement lines by number.
   - *Mod-nam* specifies that the preceding breakpoint is a breakpoint only in the named program module. The semicolon (;) is required.
   - You can specify a maximum of 10 *stmnt-break* breakpoints. If you specify more than 10 breakpoints, BASIC signals the error, "No room".

General Rules

1. If you specify a *stmnt-break* or *block* that does not exist, no break occurs, BASIC does not signal an error or warning, and the program executes normally.
2. To disable program breakpoints, use the UNBREAK command.

Examples

```
#BREAK 30.2, 500;PROGB, 2000.3;PROGC

#BREAK ON CALL

#CON

BREAK at line 30 statement 2

#
```

2.0 CONTINUE (BASIC–PLUS–2)

Function

The CONTINUE command continues program execution.

Format

```
CONTINUE
```

Syntax Rules

None

General Rules

1. When you have finished entering debugger commands, type CONTINUE to resume program execution.

Examples

```
*BREAK ON LOOP
*CON
```

3.0 CORE (BASIC–PLUS–2)

Function

The CORE command returns the number of words currently allocated in memory for your entire task. Use the CORE command in conjunction with the FREE, STRING, and I/O BUFFER commands to determine how memory is allocated for your task.

Format

```
CORE
```

Syntax Rules

None.

General Rules

1. The CORE command displays the total number of words currently allocated to your task.
2. The maximum allowable program space is 32K words on *RSX–11M/M–PLUS systems* and 31K words on *RSTS/E systems,* minus the size of your resident library. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on program space and resident libraries.
3. You can use the CORE command only when at least one program module has been compiled with the /DEBUG qualifier. Note, however, that the number returned by the CORE command reflects the memory allocation for the entire task, not just the module compiled with /DEBUG.
4. Knowing the size of core memory can help you control the size of your program and optimize accordingly. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on optimization.

Examples

```
#CORE
CORE =   7647

#
```

4.0 ERL (BASIC–PLUS–2)

Function

The ERL command returns the number of the line executing when the last error occurred.

Format

```
ERL
```

Syntax Rules

None.

General Rules

1. The ERL command tells you the number of the line executing when the last error occurred.
2. If no errors have occurred, the result returned by ERL is undefined.

Examples

```
*ERL
ERL = 1050

*
```

5.0 ERN (BASIC–PLUS–2)

Function

The ERN command returns the 1– to 6–character name of the program module that was executing when the last successfully handled error occurred. If a fatal error was not successfully trapped, control passes from the debugger to command level.

Format

```
ERN
```

Syntax Rules

None.

General Rules

1. The ERN command returns a module name only when an error has been successfully handled.
2. If no errors have occurred, the result returned by ERN is undefined.

Examples

```
#ERN
ERN$ =  CHECKS

#
```

6.0 ERR (BASIC–PLUS–2)

Function

The ERR command returns the error number of the last error that occurred.

Format

```
ERR
```

Syntax Rules

None.

General Rules

1. The ERR command tells you the number of the last error.
2. If no errors have occurred, the result returned by ERR is undefined.
3. Refer to Appendix B in *BASIC on RSTS/E Systems* or *BASIC on RSX–11M/M–PLUS Systems* for a list of errors and their numbers.

Examples

```
#ERR
ERR =  55

#
```

7.0 EXIT (BASIC–PLUS–2)

Function

The EXIT command returns control to BASIC if you are executing a program in the BASIC environment and to command level if you are executing a task-built program.

Format

```
EXIT
```

Syntax Rules

None.

General Rules

1. The EXIT command does not close open channels.

Examples

```
*EXIT
```

8.0 FREE (BASIC–PLUS–2)

Function

The FREE command returns the number of words currently available in memory for I/O and string operations before BASIC must perform another memory extension. Use the FREE command in conjunction with the CORE, STRING, and I/O BUFFER commands to determine how memory is allocated for your task.

Format

```
FREE
```

Syntax Rules

None.

General Rules

1. The FREE command returns an integer corresponding to the number of free words available in memory for I/O and string operations.
2. When string or I/O operations exceed the available free space, BASIC extends the amount of memory allocated for your task.
3. Knowing the amount of free space available can help you control the size of your program and optimize accordingly. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on optimization.

Examples

```
#FREE
FREE =   184

#
```

9.0 I/O BUFFER (BASIC–PLUS–2)

Function

The I/O BUFFER command returns the number of words currently allocated for I/O buffer space. Use the I/O BUFFER command in conjunction with the CORE, STRING, and FREE commands to determine how memory is allocated for your task.

Format

```
I/O BUFFER
```

Syntax Rules

None.

General Rules

1. The I/O BUFFER command tells you the total number of words allocated for I/O buffer space.
2. Knowing the size of the I/O buffer can help you control the size of your program and optimize accordingly. Consult *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on optimization.

Examples

```
#I/O BUFFER
I/O BUFFERS =  1765

#
```

10.0 LET (BASIC–PLUS–2)

Function

The LET command allows you to change the contents of program variables.

Format

```
                  ⎧ vbl2  ⎫
LET vbl1 =        ⎨       ⎬
                  ⎩ const ⎭
```

Syntax Rules

1. *Vbl1* specifies the numeric or string variable you want to change. If you attempt to create a new variable with the LET command, BASIC signals "Illegal syntax in LET".
2. *Const* or *vbl2* specifies the new value for *vbl1*. The LET command allows constants or variables as arguments but does not allow expressions.
3. You cannot set string variables to a null string with the LET command. If you try to do so, BASIC signals "Illegal syntax in LET". However, you can set a variable to the null string in your source program and then assign that variable to another variable with the LET debugger command. For example:

```
1000     NULL$= ""
1010     A$="HELLO"
1020     PRINT A$
```

Compile or run the program with the /DEBUG qualifier, establish a breakpoint at line 1020, and set A$ to the null string with the LET command:

```
BREAK at line 1020


*LET A$ = NULL$
```

General Rules

1. You can change only one variable with each LET command. To change more than one program variable, you must enter more than one LET command.
2. When executing a task-built program, you can change program variables only in program modules compiled with the /DEBUG qualifier.
3. You cannot access program variables across program modules. That is, you cannot access a variable in SUB1 from the main program or from another subprogram, and you cannot access a variable in the main program from a subprogram.
4. BASIC signals "Illegal syntax in LET" when you try to access a variable across modules or in a module not compiled with the /DEBUG qualifier.

LET

Examples

```
#LET A%=15%
#LET NAME$="MITCHELL"
#
```

11.0 PRINT (BASIC–PLUS–2)

Function

The PRINT command allows you to display the current contents of program variables.

Format

```
PRINT vbl
```

Syntax Rules

1. *Vbl* specifies the numeric or string variable you want to display.
2. The PRINT command does not allow constants or expressions as arguments.

General Rules

1. You can display only one variable with each PRINT command. To display more than one program variable, you must enter more than one PRINT command.
2. When executing a task-built program, you can access only those variables contained in program modules that have been compiled with the /DEBUG qualifier.
3. You cannot access variables across program modules. That is, the variable you want to display must exist in the current program module. If you try to display a variable in another program module, BASIC signals "Illegal syntax in PRINT".

Examples

```
#PRINT C
 23

#
```

12.0 RECOUNT (BASIC–PLUS–2)

Function

The RECOUNT command tells you how many characters were transferred by the last I/O operation.

Format

```
RECOUNT
```

Syntax Rules

None.

General Rules

1. The RECOUNT command tells you how many characters, including blanks and terminators, were transferred by the last input or output statement.
2. If your program has open files and reaches the end of the file before closing open channels or executing the END statement, the debugger signals "End-of-file on device". If you then try to continue program execution by typing the CONTINUE command, the debugger signals "Can't CONTINUE or STEP". When you then EXIT the debugger mode, files are not closed, and data is not transferred. If you include an error handler to pass control to the END statement, BASIC will close files and transfer data.

Examples

```
*RECOUNT
RECOUNT =  19

*
```

13.0 REDIRECT (BASIC–PLUS–2)

Function

The REDIRECT command allows you to direct all debugging I/O operations to a specified terminal.

Format

```
REDIRECT term-nam
```

Syntax Rules

1. *Term-nam* specifies the name of an unattached terminal. It must be an unquoted string that corresponds to a terminal name, or BASIC signals the error "Cannot open device".

General Rules

1. After you type the REDIRECT command in response to the debugger prompt, all debugger I/O is directed to the terminal you specify. The program executes on the terminal that issued the RUN command.
2. Use another REDIRECT command to direct debugger I/O back to the terminal on which the program is executing.
3. You can use the REDIRECT command only when at least one program module has been compiled with the /DEBUG qualifier.
4. If the specified terminal is allocated, the debugger will signal "Cannot open device" on *RSTS/E systems*. On *RSX–11M/M–PLUS systems*, the debugger stops executing until the specified terminal is available and does not signal an error.

Examples

```
*REDIRECT KB2:
```

14.0 STATUS (BASIC–PLUS–2)

Function

The STATUS command returns a word-length integer that contains information about the last opened file.

Format

```
STATUS
```

Syntax Rules

None.

General Rules

1. The debugger returns the last STATUS word.

RSX

2. Depending on the error, the STATUS word on *RSX–11M/M–PLUS systems* displays a value representing one of the following:

   - The RMS–11 primary status field (STS) or the RMS secondary status field (STV). See the *RMS–11 MACRO User's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation set by the DEV field of the FAB. See the *RMS–11 MACRO User's Guide* for more information.
   - In the event of a directive error, the Directive Status Word ($DSW) and its corresponding error code. See the *RSX–11M/M–PLUS Mini Reference* for the error codes.
   - The STATUS field of a QIO. See the *RSX–11M/M–PLUS I/O Driver's Reference Manual* for more information.
   - The first word of a GETLUN directive describing device characteristics. See the *RSX–11M/M–PLUS Executive Reference Manual* for more information.
   - See *BASIC on RSX–11M/M–PLUS Systems* for information on STATUS values set for an OPEN file operation with no errors.

(RSTS)

3. Depending on the error, the STATUS word on *RSTS/E systems* displays a value representing one of the following:

   - The RMS–11 primary status field (STS) or the RMS secondary status field (STV). See the *RMS–11 MACRO User's Guide* for more information.
   - The device characteristics after an RMS–11 OPEN file operation set by the DEV field of the FAB. See the *RMS–11 MACRO User's Guide* for more information.
   - For OPEN operations where no errors occur, the status word describes the device characteristics of the FIRQB and FQFLAG field. The first 7 bits describe the device, and bits 7 through 15 describe characteristics of the OPEN statement. See the *BASIC–PLUS Language Manual* and the *RSTS/E System Directives Manual* for more information on STATUS values.

Examples

```
#STATUS
STATUS =   31

#
```

15.0 STEP (BASIC–PLUS–2)

Function

The STEP command causes the program module to execute statement by statement, stopping after a specified number of statements have executed.

Format

```
STEP [ int-const ]
```

Syntax Rules

1. *Int-const* specifies the number of statements to be executed before the program stops. It must be a positive integer from 1 to 32767.
2. STEP with no *int-const* is the same as specifying STEP 1. Only one statement executes and the program then stops.
3. If you do not include a space between the command and the *int-const*, only one statement executes.

General Rules

1. When executing a task-built program, only statements in program modules compiled with the /DEBUG qualifier in effect are counted. If a module not compiled with the /DEBUG qualifier executes before a module compiled with the /DEBUG qualifier, the program does not stop until the specified number of statements in the module compiled with /DEBUG have executed.
2. Typing a carriage return in response to the # prompt is the same as typing STEP 1 or STEP with no *int-const*. The next statement executes and the program stops.
3. Typing a line feed in response to the # prompt has no effect. The debugger waits for a carriage return and then signals an error.

Examples

```
BREAK at line 1050 statement 1
#STEP 2
#CON
STEP at line 1050 statement 3
#
```

16.0 STRING (BASIC–PLUS–2)

Function

The STRING command tells you how many words are currently allocated for string storage. Use the STRING command in conjunction with the CORE, I/O BUFFER, and FREE commands to determine how memory is allocated for your task.

Format

```
STRING
```

Syntax Rules

None.

General Rules

1. The STRING command tells you how many words are allocated for string operations for your entire task, not just for the currently executing program module.
2. Knowing how much memory is allocated to string operations can help you control the size of your program and optimize accordingly. See *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for information on optimization.

Examples

```
*STRING
STRING =  2086

*
```

17.0 TRACE (BASIC–PLUS–2)

Function

The TRACE command displays line numbers and statement numbers as the program executes.

Format

```
TRACE
```

Syntax Rules

None.

General Rules

1. The TRACE command does not affect program execution or breakpoints.
2. When executing a task-built program, you can use the TRACE command only in program modules that have been compiled with the /DEBUG qualifier.
3. The TRACE command remains in effect until the program module finishes executing, until you specify UNTRACE after a program breakpoint, or until BASIC reaches a module not compiled with the /DEBUG qualifier. When BASIC returns to a module compiled with DEBUG, tracing resumes.

Examples

```
#TRACE
#BREAK 300
#CONT
at line 100 statement 1
at line 100 statement 2
at line 200 statement 1
BREAK at line 300 statement 1
#BREAK 500
#CONT
```

18.0 UNBREAK (BASIC–PLUS–2)

Function

The UNBREAK command disables previously set breakpoints in programs and subprograms.

Format

```
UNBREAK  { ON block                    }
         { [ ON ] stmnt-break,...      }

block:   { CALL }
         { DEF  }
         { LOOP }

stmnt-break:  lin-num [ .stmnt-num ] [ ;mod-nam ]
```

Syntax Rules

1. The ON keyword is required to disable *block* breakpoints.
2. UNBREAK with no parameters disables all previously specified *stmnt-break* breakpoints. *Block* breakpoints are not disabled.
3. *Stmnt-break* specifies a particular line number or statement where execution is to stop.

   - *Lin-num* specifies a program line.
   - *Stat-num* specifies a particular statement associated with *lin-num*. The period (.) is required and must immediately follow the line number. BASIC signals an error if you include a space between *lin-num* and *stat-num*. The listing file lists statements on multi-statement lines by number.
   - *Mod-nam* specifies that the preceding breakpoint is a breakpoint only in the named program module. The semicolon (;) is required.
   - You can disable more than one *stmnt-break* breakpoint with the UNBREAK command, but you must separate them with commas.
   - *Mod-nam* specifies a program module compiled with the /DEBUG qualifier in effect. When *mod-nam* is specified, the line number specified is disabled as a breakpoint only in the named program. If the breakpoint has not been previously set, BASIC signals an error.
   - If *lin-num* or *stat-num* do not exist, the debugger signals the error "Bad line spec in (UN)BREAK".

UNBREAK

General Rules

None.

Examples

```
#UNBREAK ON LOOP
#UNBREAK 100;GAMES, 500, 600,2
#CON
```

19.0 UNTRACE (BASIC–PLUS–2)

Function

The UNTRACE command disables the TRACE command.

Format

```
UNTRACE
```

Syntax Rules

None.

General Rules

1. Enter the UNTRACE command when the program stops executing after encountering a specified breakpoint.

Examples

```
*UNTRACE
*CON
```

Appendix A
Reserved BASIC Keywords

%ABORT
%CDD
%CROSS
%ELSE
%END
%FROM
%IDENT
%IF
%INCLUDE
%LET
%LIST
%NOCROSS
%NOLIST
%PAGE
%SBTTL
%THEN
%TITLE
%VARIANT
ABORT
ABS
ABS%
ACCESS
ACCESS%
ACTIVE
ALIGNED
ALLOW
ALTERNATE
AND
ANY
APPEND
AS
ASC
ASCII
ATN
ATN2
BACK
BASE
BEL
BINARY
BIT
BLOCK
BLOCKSIZE
BS
BUCKETSIZE
BUFFER
BUFSIZ
BY
BYTE
CALL
CASE
CCPOS
CHAIN
CHANGE
CHANGES
CHECKING
CHR$
CLK$
CLOSE
CLUSTERSIZE
COM
COMMON
COMP%
CON
CONNECT
CONSTANT
CONTIGUOUS
COS
COT
COUNT
CR
CTRLC
CVT$$
CVT$%
CVT$F
CVT%$
CVTF$
DAT
DAT$
DATA
DATE$
DECIMAL
DECLARE
DEF
DEFAULTNAME
DEL
DELETE
DESC
DET
DIF$
DIM
DIMENSION
DOUBLE
DOUBLEBUF
DUPLICATES
DYNAMIC
ECHO
EDIT$
ELSE
END
EQ
EQV
ERL
ERN$
ERR
ERROR
ERT$
ESC
EXIT
EXP
EXPLICIT
EXTEND
EXTENDSIZE
EXTERNAL
FF
FIELD
FILE
FILESIZE
FILL
FILL$
FILL%
FIND
FIX
FIXED
FLUSH
FNAME$
FNEND
FNEXIT
FOR
FORMAT$
FORTRAN
FREE
FROM
FSP$
FSS$
FUNCTION
FUNCTIONEND
FUNCTIONEXIT
GE
GET
GETRFA

GFLOAT
GO
GOBACK
GOSUB
GOTO
GROUP
GT
HFLOAT
HT
IDN
IF
IFEND
IFMORE
IMAGE
IMP
INACTIVE
INDEXED
INPUT
INSTR
INT
INTEGER
INV
INVALID
ITERATE
KEY
KILL
LEFT
LEFT$
LEN
LET
LF
LINE
LINO
LINPUT
LIST
LOC
LOCKED
LOG
LOG10
LONG
LSET
MAG
MAGTAPE
MAP
MAR
MAR%
MARGIN
MAT
MAX
MID
MID$
MIN
MOD
MOD%
MODE
MODIFY
MOVE
NAME
NEXT
NOCHANGES
NODATA
NODUPLICATES
NOECHO
NOEXTEND
NOMARGIN
NONE
NOPAGE
NOREWIND
NOSPAN
NOT
NUL$
NUM
NUM$
NUM1$
NUM2
ON
ONECHR
ONERROR
OPEN
OPTION
OR
ORGANIZATION
OTHERWISE
OUTPUT
OVERFLOW
PAGE
PEEK
PI
PLACE$
POS
POS%
PPS%
PRIMARY
PRINT
PROD$
PUT
QUO$
RAD$
RANDOM
RANDOMIZE
RCTRLC
RCTRLO
READ
REAL
RECORD
RECORDSIZE
RECORDTYPE
RECOUNT
REF
REGARDLESS
RELATIVE
REM
REMAP
RESET
RESTORE
RESUME
RETURN
RFA
RIGHT
RIGHT$
RND
ROUNDING
RSET
SCALE
SCRATCH
SEG$
SELECT
SEQUENTIAL
SETUP
SGN
SI
SIN
SINGLE
SIZE
SLEEP
SO
SP
SPACE$
SPAN
SPEC%
SQR
SQRT
STATUS
STEP
STOP
STR$
STREAM
STRING
STRING$
SUB
SUBEND
SUBEXIT
SUBSCRIPT
SUM$
SWAP%
SYS
TAB
TAN
TEMPORARY
TERMINAL
THEN
TIM
TIME
TIME$
TO
TRM$
TRN
TYP
TYPE
TYPE$
UNALIGNED
UNDEFINED
UNLESS
UNLOCK
UNTIL
UPDATE
USAGE$
USEROPEN
USING
USR$
VAL
VAL%
VALUE
VARIABLE
VARIANT
VFC
VIRTUAL
VPS%
VT
WAIT
WHILE
WINDOWSIZE
WORD
WRITE
XLATE
XOR
ZER

Appendix B
Program and Subprogram Coding Conventions

This appendix presents a suggested format for coding BASIC programs. The recommended program order and documenting procedures clarify the program's history, purpose, and logical development. This organization also helps the program to run faster and with fewer errors.

This format is by no means intended to represent the only way of coding BASIC programs. It is a sample format that can be adapted and modified to suit individual applications.

```
10  %TITLE "<module-name> - <terse functional description>"
    %SBTTL "Overall description and modification history"
    %IDENT "X00.00"
    !
    !                          COPYRIGHT (c) 1982 BY
    !              DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
    !
    ! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND  MAY  BE  USED  AND
    ! COPIED  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH
    ! THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR ANY
    ! OTHER  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAIL-
    ! ABLE TO ANY OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF THE  SOFT-
    ! WARE IS HEREBY TRANSFERRED.
    !
    ! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT  NO-
    ! TICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIP-
    ! MENT CORPORATION.
    !
    ! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY  OF
    ! ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
    !
```

```
!++
!
! FACILITY:
!
!   <Facility name>
!
! ABSTRACT:
!
!   A short 3-6 line abstract of the function of the module.  If  a
!   full functional specification can be given in 3-6 lines, replace
!   "ABSTRACT" above by "FUNCTIONAL  DESCRIPTION,"  and  delete  the
!   "FUNCTIONAL DESCRIPTION" section below.
!   !   !           [margins and tabs]                              !
!
! ENVIRONMENT:
!
!   [Pick one:]
!   PDP-11 user mode [with <Operating system> dependencies]
!   VAX-11 user mode.
!   PDP-11 and VAX-11 user mode.
!
! AUTHOR: <Your name>, CREATION DATE: <dd Mmmmmmmmm yyyy>
!
! MODIFIED BY:
!
!       <Your name>, <dd-Mmm-yy>: VERSION X00.00
! 000 - Original version of module.
!
!--
%SBTTL "Full description"

[Pick at most one of FUNCTION or SUB below.  Include parameters on
 either.  For main programs, omit FUNCTION or SUB statement and
 parameters.]
FUNCTION <datatype> <name>                                          &
SUB <name>                                                          &
   (<datatype> <param>,             ! <Description>                 &
    <datatype> <param>)             ! <Description>

!++
!
! FUNCTIONAL DESCRIPTION:
!
!   A detailed functional description of the routine.  This  should
!   detail  the  steps of the process, the use of external functions
!   and subprograms (including system services, RTL routines, SYSLIB
!   routines), and so forth.
!   !   !           [margins and tabs]                              !
!
! FORMAL PARAMETERS:
!
!   <name>,<access type><data type>,<arg mech><arg form>
!       A description of the meaning of  the  parameter,  its  legal
!       values,  etc.  Repeat for each parameter.  If a main program
!       rather  than  a  function  or  subroutine,  use  the  COMMAND
!       STRUCTURE section.
!       <access type> is m, r, or w for modify, read, or write.
!       <datatype> is b, d, g, h, l, p, s, t, or w for BYTE, DOUBLE,
!           GFLOAT,  HFLOAT,  LONG,  packed  (DECIMAL), SINGLE, text
!           (STRING), or WORD.
!       <arg mech> is d, r, or v for BY DESC, BY REF, or BY VALUE.
!       <arg form> is <null> or a for scalar or array.
!   !   !   !       [margins and tabs]                              !
!
```

```
! IMPLICIT INPUTS:
!
!   Describe all uses of the values of global storage  objects  used
!   by the routine.
!   !   !           [margins and tabs]                                 !
!
! IMPLICIT OUTPUTS:
!
!   Describe all modifications to the values of global  storage  ob-
!   jects used by the routine.
!   !   !           [margins and tabs]                                 !
!
! FUNCTION VALUE:
! COMPLETION CODES:
!
!   If a function, describe the value returned.  If  the  value  re-
!   turned  is  a  status  indicator, use COMPLETION CODE and delete
!   FUNCTION VALUE; if the result of some computation, use FUNCTION
!   VALUE and delete COMPLETION CODE.
!   If a SUB, delete FUNCTION VALUE and enter "None."
!   !   !           [margins and tabs]                                 !
!
! SIDE EFFECTS:
!
!   Describe all functional side effects that are not  evident  from
!   the  invocation interface.  This includes changes in storage al-
!   location, process status, file operations (including the command
!   terminal), errors signalled, etc.
!   !   !           [margins and tabs]                                 !
!
!--
%SBTTL "Declarations"
!
! ENVIRONMENT SPECIFICATION:
!
OPTION                                                                 &
    <option clause>,                                                   &
    <option clause>
!
! DATATYPE SPECIFICATION:
!
RECORD <name>                              ! <Description>
    <record declaration>
END RECORD
!
! INCLUDE FILES:
!
%INCLUDE "<Filespec>"
!
! EQUATED SYMBOLS:
!
DECLARE <datatype> CONSTANT                                            &
    <name> = <value>,                      ! <Description>             &
    <name> = <value>                       ! <Description>
!
! LOCAL STORAGE:
!
DECLARE                                                                &
    <datatype>                                                         &
        <name>,                            ! <Description>             &
        <name>,                            ! <Description>             &
    <datatype>                                                         &
        <name>,                            ! <Description>             &
        <name>                             ! <Description>
!
```

```
! GLOBAL STORAGE:
!
COMMON (<name>)                         ! <Description>                 &
    <datatype>                                                          &
        <name>,                         ! <Description>                 &
        <name>,                         ! <Description>                 &
    <datatype>                                                          &
        <name>,                         ! <Description>                 &
        <name>                          ! <Description>
MAP (<name>)                            ! <Description>                 &
    <datatype>                                                          &
        <name>,                         ! <Description>                 &
        <name>,                         ! <Description>                 &
    <datatype>                                                          &
        <name>,                         ! <Description>                 &
        <name>                          ! <Description>
!
! EXTERNAL REFERENCES:
!
EXTERNAL <datatype> CONSTANT                                            &
    <name>                              ! <Description>
EXTERNAL <datatype>                                                     &
    <name>                              ! <Description>
EXTERNAL <datatype> FUNCTION                                            &
    <name>                              ! <Function description>        &
       (<datatype> BY <mech>,           ! <Argument description>        &
        <datatype> BY <mech>)           ! <Argument description>
EXTERNAL SUB                                                            &
    <name>                              ! <Function description>        &
       (<datatype> BY <mech>,           ! <Argument description>        &
        <datatype> BY <mech>)           ! <Argument description>
!
! INTERNAL REFERENCES:
!
DECLARE <datatype> FUNCTION                                             &
    <name>                              ! <Function description>        &
       (<datatype>,                     ! <Argument description>        &
        <datatype>)                     ! <Argument description>
    %SBTTL "Environment initialization"
    !+
    !   Set up global error handler
    !-
    ON ERROR GO TO 31000
    [Alternately, local error handlers can be set up where needed.]
    %SBTTL "<Major section name>"
<Major section name>:
    !+
    !   <Section description>
    !-
    [Repeat once for each major section.]
    %SBTTL "Internal subroutine: <symbolic name>"
    [Access via GOSUB <symbolic name>]
<symbolic name>:
    !+
    !
    ! FUNCTIONAL DESCRIPTION:
    !
    ! IMPLICIT INPUTS:
    !
    ! IMPLICIT OUTPUTS:
    !
```

```
        ! SIDE EFFECTS:
        !
        !-
        RETURN
        %SBTTL "Internal function - <name>"
        [Access via <name> (<params>)]
        DEF <datatype> <name>                                                  &
           (<datatype> <name>,          ! <Description>                        &
            <datatype> <name>)          ! <Description>
        !+
        !
        ! FUNCTIONAL DESCRIPTION:
        !
        ! FORMAL PARAMETERS:
        !
        ! IMPLICIT INPUTS:
        !
        ! IMPLICIT OUTPUTS:
        !
        ! FUNCTION VALUE:
        !
        ! SIDE EFFECTS:
        !
        !-
        END DEF
        %SBTTL "RSTS/E CCL entry point"
        [This section is for RSTS/E CCL's only]
30000   !+
        !   CCL entry point:
        !-
        %SBTTL "Common error handling"
31000   !+
        !   Common error handling:
        !-
        %SBTTL "Module end"
32767   END <FUNCTION, null, or SUB>
```

Index

This index provides a complete cross-reference to the information in this manual. In the index the following convention is used:

Example	Explanation
12t	A page number followed by a *t* indicates a table.

For material not covered in this manual, see the Master Index in the back of the *BASIC Reference Manual*. The Master Index contains a list of the major references to information throughout the BASIC documentation set.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Index

The Master Index contains a list of the major references to subjects in the *BASIC Reference Manual*, the *BASIC User's Guide*, and the system-specific manuals. The index uses the following conventions:

Example	Explanation
1–8t	A page number followed by a *t* indicates a table.
4–36f	A page number followed by an *f* indicates a figure.

Entries in the Master Index are also preceded by an acronym indicating which manual the page number refers to:

Acronym	Title
LM	*BASIC Reference Manual*
UG	*BASIC User's Guide*
RSTS	*BASIC on RSTS/E Systems*
RSX	*BASIC on RSX Systems*
VMS	*BASIC on VAX/VMS Systems*

Where a subject references more than one manual, references to the *BASIC Reference Manual* appear first, the *BASIC User's Guide* second, and the system-specific manuals appear last in alphabetical order.

For a more complete list of references, see the individual indexes in the back of each manual.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**

In Canada call **800–267–6146**

In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary:
809–754–7575

BASIC
Reference Manual
AA–L334A–TK
AD–L334A–T1

Reader's Comments

Note: This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement._____

Did you find errors in this manual? If so, specify the error and the page number._____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code or Country _____

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: BSSG Publications ZKO1–3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, N.H. 03062

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line