BASIC
User's Guide

Order No. AA–L335A–TK
Including AD–L335A–T1

February 1984

This manual describes the use of BASIC–PLUS–2 on PDP–11 systems and VAX BASIC on VAX/VMS systems.

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

Chapter 3 Program Control

Chapter 4 Strings

Chapter 5 Data Definition

Chapter 6 Functions

Chapter 7 Arrays

Chapter 8 Formatting Output with the PRINT USING Statement

Chapter 9 RMS Files

Index

Figures

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 features and use of the BASIC language on VAX/VMS, RSTS/E, and RSX–based systems. It is intended to be used with the *BASIC Reference Manual, BASIC on VAX/VMS Systems, BASIC on RSTS/E Systems,* and *BASIC on RSX–11M/M–PLUS Systems.*

Intended Audience

This manual is intended for programmers familiar with computer concepts and the BASIC language.

Document Structure

This manual has eleven chapters and two appendixes.

Chapter 1	Describes BASIC language elements and writing BASIC programs.
Chapter 2	Describes simple BASIC input and output.
Chapter 3	Describes ways to control the execution of BASIC programs.
Chapter 4	Describes strings and how to modify them.
Chapter 5	Describes how to define data with declarative statements.
Chapter 6	Describes BASIC library functions and user-defined functions.
Chapter 7	Explains how to create and use arrays.
Chapter 8	Explains how to format program output with the PRINT USING statement.
Chapter 9	Describes file I/O.
Chapter 10	Explains how to use compiler directives.
Chapter 11	Explains how to handle errors.
Appendix A	Lists the BASIC reserved keywords.
Appendix B	Contains a programming template.

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.

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 (map)
- COMMON (com)
- FUNCTION (func)
- DEF (def)
- SUB (sub)

Others are abbreviated forms of words. For example:

- Variable (vbl)
- Unsubscripted (unsubs)
- Subscripted (subs)
- String (str)
- Constant (const)
- Expression (exp)
- Name (nam)
- Conditional (cond)
- Integer (int)
- File-specification (file-spec)
- Data-type (data-type)

Most mnemonics used in formats are combinations of mnemonics:

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

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

When you look at 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 further defined in the lower portion of the box, like this:

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

From this portion of the format, you can see that the data-type keyword must be followed by an array name or a simple variable name. The portion of the format in brackets indicates that you can specify another data-type keyword and another array name or simple variable name. The comma and ellipsis (...) 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. In most cases, BASIC signals an error if the 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 in the following two tables. Less frequently used mnemonics and most complex mnemonics are defined as they occur in syntax formats.

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; Table 5–1 in this manual lists and describes BASIC data-type keywords
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 to Version 2 of *PDP–11 BASIC–PLUS–2* as implemented on *RSTS/E*, *RSX–11M*, and *RSX–11M–PLUS* systems.
Cannot	Cannot indicates that 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 subprogram is 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.

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

Chapter 1
Elements of a BASIC Program

This chapter describes the elements that make up a BASIC program.

1.1 Line Numbers and Alphanumeric Labels

A BASIC program is a series of instructions for the BASIC compiler. These instructions are in the form of BASIC statements. BASIC programs require at least one line number at the first statement of the program.

BASIC 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 and to 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. Embedded spaces, tabs, and commas are invalid.

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

- Line numbers
- Statements (optionally preceded with alphanumeric labels)
- Comment fields
- Line terminators (carriage return)

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*, or 132 characters in *BASIC–PLUS–2* on *RSX–11M/M–PLUS systems*.

Program lines contain the BASIC keywords, operators, and operands that make up a BASIC program. If the line you type in response to the prompt starts with a number in the first column, BASIC treats it as a program line — that is, as part of the current source program. BASIC stores this text in ascending line-number order, and if the line you type has the same number as an existing line, the new line replaces the old one.

If a program line is too long for one line of text, you can continue the line by typing an ampersand (&) as the last character before the carriage return (RET).

Several statements can be associated with a single line number. If these statements are on one line, they 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 this line number refers to all three statements. In the preceding example, BASIC cannot execute just one of the statements without executing the other two.

You can extend a multi-statement program line onto several lines by ending each continued line with an ampersand (&) and beginning each continuation line with a backslash. For example:

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

However, programs written in this format tend to be cluttered and hard to read: BASIC now allows you to identify a continued line by placing a space or tab at the beginning of each continuation line. BASIC assumes such a line is a continuation of the preceding line, and, unless it is part of an IF–THEN–ELSE statement, assumes it begins a new statement. For example:

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

A single statement that spans several text lines still requires an ampersand at the end of each continued line. For example:

```
100     OPEN "SAMPLE.DAT" AT FILE #2%,     &
        SEQUENTIAL VARIABLE,               &
        RECORDSIZE 80%
```

The ampersand must come immediately before the carriage return.

The IF–THEN–ELSE construction requires that BASIC interpret this new format somewhat differently. If a continuation line begins with THEN or ELSE, no statement separator is assumed. Similarly, in a line following a THEN or ELSE, there is no implied statement separator. For example:

```
100     IF (A$ = "YES") OR (A$ = "Y")
        THEN
           PRINT "The user gave a positive response"
        ELSE
           PRINT "The user gave a negative response"
        END IF
```

Because a leading space or tab implies a continuation line, compiler commands and immediate mode statements must begin in column one with no leading spaces or tabs. If you enter a compiler command or immediate mode statement, no further continuation lines can be added to that program line.

Note

> You can continue any BASIC statement. However, a statement following a REM or DATA statement must have a new line number. This is because all text between the keyword REM and the next line number is ignored. Similarly, all text between the keyword DATA and the next line number is treated as data.

The EDIT command treats the new format continuation lines as normal continuation lines. Thus, you can edit these continuation lines by using the sub-line clause of the EDIT command.

A label is a 1– to 31–character identifier used to logically identify a statement or block of statements. You can reference a label anywhere you can reference a line number, with two exceptions: 1) you cannot compare the ERL variable (the line on which the last error occurred) with a label, and 2) you cannot use a label as the argument to the RESUME statement. A label cannot begin in the first character position on a line; only line numbers and immediate mode statements can begin in the first column.

When used to define a block of statements, the label must end with a colon. However, when you use a label to refer to a block of statements (for example, in a GOSUB or GOTO statement), you do not specify a colon. For example:

```
100     RANDOMIZE
        NUMBER% = INT(9% * RND) + 1%
   Ask:
        INPUT "A number between 1 and 10 please"; A%
        GOTO Win IF A% = NUMBER%
        PRINT "Try again"
        GOTO Ask
   Win:
        PRINT "YOU WIN!!!"
        END
```

Note that the labels themselves end with a colon, but the references to them do not. Any BASIC statement can be preceded by one or more labels.

1.2 BASIC Character Set

BASIC uses the full ASCII character set. This includes:

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

Appendix C of *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 letters within quotation marks or within a DATA statement)
- Does not process characters in REMARK statements or comment fields

You can use nonprinting characters in your program, for example, in string constants, but to do so you must either: 1) use a predefined constant such as ESC and DEL or 2) use the CHR$ function to specify an ASCII value. See Section 1.5.4 for more information on predefined constants. See Chapter 6 for more information on the CHR$ function.

For user-supplied names, BASIC makes no distinction between upper- and lowercase letters. This means that MY_VBL and my_vbl always refer to the same variable.

1.3 Keywords

A keyword is an element of the BASIC language. Keywords are reserved words because the program uses them to:

- Define data
- Perform operations
- Invoke functions

Therefore, keywords cannot be used as variable names, nor as names for MAP or COMMON areas. Appendix A contains a list of BASIC reserved words.

Keywords determine whether the statement is executable or nonexecutable:

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

Every statement except LET and null statements must begin with a keyword. A BASIC keyword cannot have embedded spaces or be split across lines of text. There must be a space or tab between the keyword and any other variables or operators.

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

Table 1–1: Keyword Space Requirements

Optional Space	Mandatory Space	No Space
GO TO	BY DESC	FNEND
GO SUB	BY REF	FNEXIT
ON ERROR	BY VALUE	FUNCTIONEND
	END DEF	FUNCTIONEXIT
	END FUNCTION	NOECHO

(continued on next page)

Table 1–1: Keywork Space Requirements (Cont.)

Optional Space	Mandatory Space	No Space
	END GROUP	NOMARGIN
	END IF	SUBEND
	END RECORD	SUBEXIT
	END SELECT	
	END SUB	
	EXIT DEF	
	EXIT FUNCTION	
	EXIT SUB	
	INPUT LINE	
	MAT INPUT	
	MAT LINPUT	
	MAT PRINT	
	MAT READ	

1.4 Program Documentation

Documentation clarifies and explains source program structure. In BASIC, there are two ways to include comments in a program: the REM statement and the comment field.

1.4.1 REM Statements

The REM statement has the format:

```
REM text
```

where:

text Is a comment.

REM statements do not affect program execution; BASIC ignores all characters between the keyword REM and the next line number. Therefore, a REM statement should be either the only statement on a line, or the last statement on a multi-statement line. For example:

```
10 REM THIS IS AN EXAMPLE

20      A=5
        B=10
        REM A EQUALS 5, B EQUALS 10
```

You can use the line number of a REM statement in a reference from another statement (for example, GOTO). However, because REM is nonexecutable, BASIC transfers control to the next executable statement following the referenced line. Because BASIC treats all text between the REM keyword and the next line number as a comment, DIGITAL recommends using comment fields for program documentation.

1.4.2 Comment Fields

A comment field is similar to the REM statement; it does not affect program compilation or execution. A comment field begins with an exclamation point (!) and ends with a carriage return. For example:

```
10        A = B + C ! This is a test.RET
20        !Assign values to variablesRET
          A = 3 !A equals 3RET
          B = 4 !B equals 4RET
          C = 5 !C equals 5RET
```

The advantage to using comment fields is that you can mix program documentation with executable statements in the same block of code. Note that comment fields in DATA statements are invalid; the compiler treats the comments as additional data.

Note

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

1.5 Constants

A constant is a quantity with a fixed value.

BASIC constants are:

- Floating-point numbers
- Integers
- Strings (ASCII characters enclosed in quotation marks)
- Packed decimal numbers (VAX–11 BASIC only)

BASIC also supplies predefined constants for ease in representing some ASCII characters and mathematic values. See Section 1.5.4 for more information on predefined constants. See Chapter 5 for more information on packed decimal numbers.

In addition, BASIC provides a special form of constant called an explicitly formed literal. See Section 5.5.3 for more information on explicitly formed literals.

1.5.1 Floating-point Constants

A floating-point constant is one or more decimal digits, optionally preceded by a plus sign or minus sign, with an optional decimal point. For example:

123.84103	–377	–12345
6.64	.005	8.0003
–9.4177	6562	25

BASIC accepts floating-point constants in the approximate range $1 * 10^{-38}$ to $1 * 10^{38}$. A floating-point constant outside this range returns a fatal error message. (VAX–11 BASIC lets you use floating-point constants in the range $.84 * 10^{-4932}$ through $.59 * 10^{4932}$. See Chapter 5 for information about GFLOAT and HFLOAT numbers.)

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

± number E ± n

where:

+ or –	Is the number's sign. The plus sign is optional, but the minus sign is mandatory for negative numbers.
number	Is the number carried to a maximum of: 1) 6 decimal places for SINGLE numbers, 2) 16 decimal places for DOUBLE numbers, 3) 15 decimal places for GFLOAT numbers (VAX–11 BASIC only), and 4) 33 decimal places for HFLOAT numbers (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). It can be 0, but not blank.

Table 1–2 compares numbers in standard and E notation.

Table 1–2: 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

1.5.2 Integer Constants

In BASIC, an integer constant is one or more decimal digits (a whole number) followed by a percent sign. For example:

29%	–8%
3432%	1%
12345%	205%

Allowable values for integer constants depend on the default data type. See Chapter 5 for more information.

Note

Unless you change the default data type, BASIC assumes that numeric constants are floating-point numbers. Thus, BASIC must convert numeric constants when assigning them to integer variables. This means your program takes slightly longer to compile. You can prevent this conversion step by using percent signs for integer constants. Note that 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).

1.5.3 String Constants

A string constant, also called a string literal, is a series of ASCII 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 ('"text"') and vice versa ("'text'"). BASIC does not accept incorrectly paired quotation marks, however. For example, these are valid strings:

"THE RECORD NUMBER DOES NOT EXIST."

'"The terminating condition has been reached."'

"Report 543"

These are not:

"QUOTATION MARKS DO NOT MATCH'

"NO DELIMITER

BASIC does not print the delimiting quotation marks when executing the program. For example:

```
10      PRINT "End-of-file reached"
20      END

RUNNH

End-of-file reached
```

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

```
10       PRINT 'Failure condition: "Record Length"'
20       END

RUNNH

Failure condition: "Record Length"
```

Characters in string constants can be letters, numbers, spaces, tabs, or any ASCII character except the string delimiter, nulls (ASCII code 0) and NAKs (ASCII code 21). BASIC determines the string constant's value 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"

When processing text between string delimiters, BASIC stores the following characters exactly as you type them:

- Lowercase letters
- Leading, trailing, and embedded spaces
- Tabs
- Special characters

However, a string literal cannot contain a null (ASCII code 0) or a NAK (ASCII code 21). If you need to create a string containing a null, you should use the CHR$ function or an explicitly formed literal with the concatenation operator. See Chapter 5 for more information about explicitly formed literals. See Chapter 6 for more information about the CHR$ function.

1.5.4 Predefined Constants

Predefined constants are symbolic representations of either: 1) ASCII characters or 2) mathematic values. They:

- Format program output to improve clarity
- Make source code easier to understand
- Let you use nonprinting characters without having to look up the ASCII values

Table 1–3 lists the predefined constants, their decimal ASCII values and purposes.

Table 1–3: 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
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 (3.14159 in single precision)

The effect of these characters depends on the device to which they are sent. For example:

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

RUNNH

NAME:
```

If you run the following program on a hardcopy terminal, BASIC prints "NAME:", backspaces to the beginning of the line, and prints five underscores.

You can also create your own named constants with the DECLARE statement. See Chapter 5 for more information about DECLARE.

1.6 Variables

A variable is a named quantity whose value can change during program execution. Each variable name refers to a unique location in the program's storage. Each location can hold only one value at a time. When using variables for calculations, BASIC always uses the most recently assigned value.

This section describes the use of implicit variables, that is, variables that are not explicitly named in declarative statements such as DECLARE, MAP, and COMMON. You can use the /TYPE=EXPLICIT qualifier to specify that all variables in the program must be explicitly declared. For more information on /TYPE=EXPLICIT and on explicit variables in general, see Chapter 5.

BASIC accepts these general types of variables:

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

See Chapter 9 for more information about RFA variables. See Chapter 5 for more information about packed decimal variables.

All variables can have subscripts that indicate their position in an array. The variable types described in the next three sections are implicit; that is, the variable name defines its data type. Integer variables end with a percent sign, string variables end with a dollar sign, and floating-point variables end with neither a percent nor dollar sign.

For new program development, DIGITAL recommends using BASIC's explicit data-typing features. See Chapter 5 for more information.

1.6.1 Floating-point Variables

A floating-point variable is a named location that stores a single floating-point value. The names of floating-point variables have a single letter followed by up to 30 letters, digits, underscores, and

periods. Variable names cannot contain embedded spaces. Therefore, the maximum length of a floating-point variable name is 31 characters:

1 letter
30 optional characters

For example:

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

If an integer value is assigned to a floating-point variable, BASIC converts the value to a floating-point number.

1.6.2 Integer Variables

An integer variable is a named location that stores a whole number.

Integer variable names have a single letter followed by up to 29 optional letters, digits, underscores, and periods. Unless the variable is explicitly declared, the name ends with a percent sign (%). Therefore, the maximum length of an integer variable name is 31 characters:

1 letter
29 optional characters
1 percent sign (%)

For example:

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

Variable names cannot contain embedded spaces.

If you assign a floating-point value to an integer variable, BASIC truncates the fractional portion of the value. It does not round to the nearest integer. In this example:

```
10 B% = -5.7
```

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

For new program development, DIGITAL recommends using BASIC's explicit data-typing features. See Chapter 5 for more information.

1.6.3 String Variables

A string variable is a named area that stores strings. String variable names have a letter followed by up to 29 optional letters, digits, underscores, and periods. The name ends with a dollar sign ($). Therefore, the maximum length of a string variable name is 31 characters:

1 letter
29 optional characters
1 dollar sign ($)

For example:

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

Variable names cannot contain embedded spaces.

Strings have both value and length. BASIC sets all string variables to a default length of zero before execution, except those in a COMMON, MAP, or virtual array. 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 PDP–11 BASIC–PLUS–2).

For new program development, DIGITAL recommends using BASIC's explicit data-typing features. See Chapter 5 for more information.

1.6.4 Subscripted Variables

A subscripted variable is a floating-point, integer, packed decimal (VAX–11 BASIC only), RFA or string variable that is part of an array. Subscripts define the variable's position in the array. When an array is first declared (in a DIMENSION, MAP, COMMON, or DECLARE statement), the bounds determine the size of the entire array. Thus, the bounds of the array define the maximum value for a subscript of that array.

On VAX/VMS systems, subscripts can be any positive integer value from 0 to 32767 in WORD mode, or 0 to 2147483647 in LONG mode. On PDP–11 systems, subscripts can be any positive integer value from 0 to 32767.

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.

In BASIC, arrays are zero-based. That is, the number of elements in any dimension always includes element number zero. For example, if you dimension an array with (5,5), the array contains 36 elements. This is because BASIC always allocates row and column zero. See Chapter 7 for more information on arrays.

1.6.5 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.

Note

In PDP–11 BASIC–PLUS–2, variables in a MAP statement that is 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. Also, in PDP–11 BASIC–PLUS–2, you can use MACRO–11 routines to initialize MAP and COMMON areas. See *BASIC on RSTS/E Systems* or *BASIC on RSX–11M/M–PLUS Systems* for more information.

1.7 Expressions

Expressions are operands (numbers, strings, constants, variables, functions, or array elements) separated by:

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

Parentheses can appear in expressions to group operands and operators, thus changing the order of evaluation.

These operators can produce:

- Numeric expressions
- String expressions
- Conditional expressions

The following sections explain these types of expressions.

1.7.1 Numeric Expressions

Numeric expressions are floating-point or integer operands separated by arithmetic operators and optionally grouped by parentheses. See Table 1–4.

Table 1–4: 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

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

A% + B% is an integer expression.

G3 * M5 is a floating-point expression.

An operation on an integer and a floating-point quantity generates a floating-point value:

B% * A is a floating-point expression.

6.8 * 5% is a floating-point expression.

Assigning a value of one data type to a variable of a different data type changes the value to the variable's data type. The following example assigns the value 32 to the integer variable A%:

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

This is called numeric conversion. See Chapter 5 for a more complete discussion of this topic.

In general, two arithmetic operators cannot occur consecutively in the same expression. The exceptions are the unary plus and unary minus. For example:

A*+B is valid.

A*–B is valid.

A*(–B) is valid.

A * + – + – B is valid.

A–*B is not valid.

1.7.2 String Expressions

String expressions are strings separated by the plus sign (+) or combinations of string functions. When it appears in a string expression, the plus sign is the string concatenation operator, not the numeric addition operator.

For example:

```
10 C$ = "THE RECORD FORMAT IS" + " SEQUENTIAL VARIABLE"
20 PRINT C$
30 END

RUNNH

THE RECORD FORMAT IS SEQUENTIAL VARIABLE
```

See Chapter 4 for more information on strings.

1.7.3 Conditional Expressions

Conditional expressions are useful for coding decision points in your program. Program control structures such as IF–THEN–ELSE, UNTIL, and WHILE all use conditional expressions to determine the points to which control is transferred.

A conditional expression contains operands and relational or logical operators. A conditional expression can contain relational expressions, logical expressions, or a combination of these. However, no matter how many operands and operators a conditional expression contains, the value it returns is either zero (false) or minus one (true).

Relational operators express comparisons between two numbers or two strings. You can compare numeric values with numeric values, and string values with string values, but you cannot compare numeric values with string values.

Logical operators express logical or Boolean relationships. A logical operator performs its operation bit-by-bit between two integer values. You can use a logical expression to test a condition, for example:

```
10      IF (A% AND B%) THEN PRINT "TRUE" ELSE PRINT "FALSE"
```

When using logical expressions to test conditions, the logical expression is false if zero and true for all other values.

The following sections describe types of conditional expressions.

1.7.3.1 Numeric Relational Expressions

Numeric relational operators compare the values of two operands and return an integer: 1) minus one if the relation is true, or 2) zero if the relation is false. For example:

Example 1

```
10      A = 10
        B = 10
        C = 15
        PRINT "RELATIONSHIP IS TRUE" IF (A <> C)
        PRINT "RELATIONSHIP IS FALSE" IF NOT (A <> C)
        END

RUNNH

RELATIONSHIP IS TRUE
```

Example 2

```
10      A = 10
        B = 10
        C = 15
        PRINT "RELATIONSHIP IS TRUE" IF (A = C)
        PRINT "RELATIONSHIP IS FALSE" IF NOT (A = C)
40      END

RUNNH

RELATIONSHIP IS FALSE
```

Table 1–5 lists numeric relational operators.

Table 1–5: 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.

1.7.3.2 String Relational Expressions

String relational operators compare string values. BASIC uses the ASCII character collating sequence to determine the relative character values. It compares the strings character by character, left to right, until it finds a difference in ASCII value. Note that the relational operator = = has a different meaning when applied to strings than when applied to numbers. For example:

```
10      A$ = "ABC"
        B$ = "ABZ"
        PRINT "ABC is less than ABZ" IF A$ < B$
        PRINT "Strings are identical." IF A$ == B$
        PRINT "ABC is greater than ABZ" IF A$ > B$
        END

RUNNH

ABC is less than ABZ
```

BASIC compares A$ and B$ character by character. The strings are identical up to the third character. Because the ASCII value of "Z" is greater than that of "C", A$ is less than B$.

If two strings 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. It then compares the remaining characters in the longer string against these spaces. For example:

```
10      A$ = "ABCDE"
        B$ = "ABC"
        PRINT "B$ COMES BEFORE A$" IF B$ < A$
        PRINT "A$ COMES BEFORE B$" IF A$ < B$
        END

RUNNH

B$ COMES BEFORE A$
```

In this program, BASIC compares "ABCDE" to "ABC " to determine which string comes first in the collating sequence.

Table 1–6 lists string relational operators and their meanings.

Table 1–6: 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.

1.7.4 Logical Expressions

A logical expression contains either: 1) an optional unary logical operator and one operand or 2) two operands separated by a logical operator. Logical expressions are valid only when the operands are integers.

BASIC determines whether the logical expression is true or false as shown in Table 1–7.

Table 1–7: 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 1–8 summarize the results of these logical operations; zero is false, minus one is true:

Table 1–8: 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.

BASIC generally accepts any nonzero value as true. However, logical operators return valid results only when minus one is specified for true values and zero for false. For example:

Example 1

```
10      A% = 2%
        B% = 4%
        PRINT "A% is true" IF A%
        PRINT "B% is true" IF B%
        IF (A% AND B%)
        THEN
            PRINT "A% AND B% is true"
        ELSE
            PRINT "A% AND B% is false"
        END IF
        END

RUNNH


A% is true
B% is true
A% AND B% is false
```

Example 2

```
10      A% = -1%
        B% = -1%
        PRINT "A% is true" IF A%
        PRINT "B% is true" IF B%
        IF (A% AND B%)
        THEN
            PRINT "A% AND B% is true"
        ELSE
            PRINT "A% AND B% is false"
        END IF
60 END

RUNNH

A% is true
B% is true
A% AND B% is true
```

In the first program, the values of A% and B% test as true. However, the logical AND of these two variables returns an unexpected result. In the second program, variables A%, B%, and the logical expression A% AND B% test as true.

The first 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 of these values tests as true because they are nonzero. However, the result of an AND operation on these two values is zero:

0 0 0 0 0 0 0 0

This is because the AND operation sets a bit in the result only if the corresponding bit is set in both operands. This value tests as false. The binary representation of minus one is:

1 1 1 1 1 1 1 1

(The leftmost bit is a sign bit.) The result of –1% AND –1% is –1%, and this value tests as true.

1.7.5 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 evaluation.

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

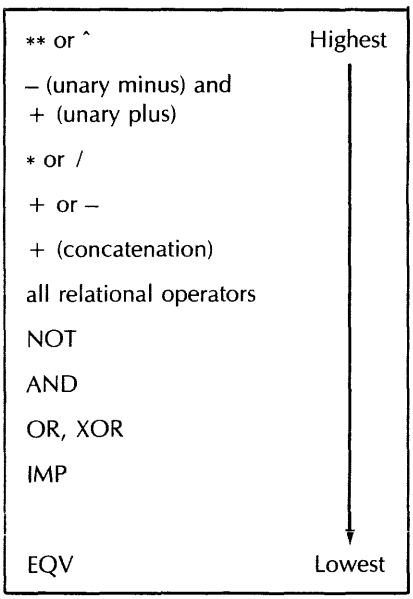
B = (25 * (16 ^(3/2)))

This expression is evaluated to 1600. Because (3/2) is the innermost parenthetical expression, BASIC evaluates it first, then (16^1.5), and finally (25 * 64).

Table 1–9 lists all operators as BASIC evaluates them. Note that:

- Operators on the same line in the table are evaluated left-to-right.
- The operators + and * are evaluated in algebraically correct order.
- BASIC evaluates A^B^C as (A^B)^C.
- BASIC evaluates expressions enclosed in parentheses first, even when the operator in parentheses has a lower precedence than operators outside the parentheses.

Table 1–9: Numeric Operator Precedence



BASIC evaluates the following expression in five steps:

A = 15^2+12^2–(35 * 8)

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

Chapter 2
Simple Input and Output

This chapter explains how to use the BASIC statements that move data to and from your program.

2.1 Program Input

BASIC programs receive data in three ways:

- You can enter data interactively while the program runs. You do this with INPUT, INPUT LINE, and LINPUT statements.
- If you know all the information your program will require, you can enter it as you write the program. You do this with READ, DATA, and RESTORE statements.
- You can read data from files outside the program. You do this with INPUT #, INPUT LINE #, and LINPUT # statements.

The following sections describe the first two types of program input. See Section 2.3 for more information about simple I/O to and from files.

2.1.1 Providing Input Interactively

The INPUT, INPUT LINE, and LINPUT statements prompt you for data while the program runs.

2.1.1.1 INPUT Statement

The INPUT statement has the format:

INPUT [prompt $\left\{ \begin{matrix} , \\ ; \end{matrix} \right\}$] vbl [,[prompt $\left\{ \begin{matrix} , \\ ; \end{matrix} \right\}$] vbl] . . .

where:

prompt — Is a string literal that clarifies the request for data. Enclose string literals in quotation marks. You can specify a separate prompt for each variable in the list.

, and ; Are punctuation marks that determine the cursor position after BASIC prints the prompt.

vbl, vbl Is a list of variables assigned values by the INPUT statement. Separate the variables with commas or semicolons.

The prompt can clarify the input request by specifying the type and number of data elements required by the program. This is especially useful when: 1) the program contains many variables or 2) someone else is running your program. For example:

```
10      INPUT "PLEASE TYPE 3 INTEGERS" ;B% ,C% ,D%
        A% = B% + C% + D%
        PRINT "THEIR SUM IS"; A%
        END

RUN

PLEASE TYPE 3 INTEGERS? 25,50,75(RET)
THEIR SUM IS 150
```

When your program runs, BASIC stops at each INPUT, LINPUT, or INPUT LINE statement and:

- Prints a string prompt, if specified, and a question mark (?) followed by a space.
  - If you have a semicolon separating the input prompt from the variable, BASIC prints the question mark and space immediately after the input prompt.
  - If you have a comma separating the input prompt from the variable, BASIC prints the input prompt, skips to the next print zone, and then prints the question mark and space.
- Waits for you to type a value for the variable named in the INPUT statement.

See Section 2.2.1 for more information about print zones.

You must provide one value for each variable in the INPUT request. If you do not, BASIC prompts again:

```
10 INPUT A,B
20 END

RUN

? 5(RET)
? 6
```

BASIC interprets a carriage return (null input) as a zero value for numeric variables and as a null string for string variables. For example:

```
? 5(RET)
? (RET)
```

These responses assign the value five to variable A and zero to variable B. In contrast, if you provide more values than there are variables, BASIC ignores the excess. In this example:

```
10 INPUT A,B,C
15 PRINT A,B,C
20 END

RUN

? 5,6,7,8(RET)

 5             6             7
```

BASIC ignores the extra value (8). Note that you can type multiple values if they are separated by commas. Because commas separate variables in line 15, BASIC prints each variable at the start of a print zone.

If you name a numeric variable in an INPUT statement, you must supply numeric data. If you supply string data to a numeric variable, BASIC signals "Illegal number" (ERR=52). If you supply a floating-point number for an integer variable, BASIC signals "Data format error" (ERR=50).

If you name a string variable in an INPUT statement, you can supply either numbers or letters, but BASIC treats the data you supply as a string. It is important to note that digits and a decimal point are also valid ASCII characters. For example:

```
10      INPUT "Please type a number"; A$
        PRINT A$

RUNNH

Please type a number? 25.5
25.5
```

BASIC interprets the response as a 4–character string.

You can type strings with or without quotation marks. However, if you want to input a string containing a comma, you should enclose the string in quotation marks. If you do not, BASIC treats the comma as a delimiter and assigns only part of the string to the variable. If you use quotation marks, be sure to type both beginning and ending marks. If you forget the end quotation mark, BASIC signals "Data format error" (ERR=50).

2.1.1.2 INPUT LINE and LINPUT Statements

The INPUT LINE and LINPUT statements prompt you for string data while your program runs. You can respond with strings that contain commas, semicolons, and quotation marks — characters the INPUT statement interprets as delimiters. The formats of these statements are:

INPUT LINE [prompt $\left\{\begin{matrix},\\;\end{matrix}\right\}$] str-vbl [,[prompt $\left\{\begin{matrix},\\;\end{matrix}\right\}$] str-vbl] . . .

LINPUT [prompt $\left\{\begin{matrix},\\;\end{matrix}\right\}$] str-vbl [,[prompt $\left\{\begin{matrix},\\;\end{matrix}\right\}$] str-vbl] . . .

where:

prompt Is a string literal that clarifies the request for data.

, and ; Are punctuation marks that determine the cursor position after BASIC prints the prompt.

str-vbl Is a string variable.

When your program runs, BASIC stops at each LINPUT or INPUT LINE statement and:

- Prints a string prompt, if specified, and a question mark (?) followed by a space.
  - If you have a semicolon separating the input prompt from the variable, BASIC prints the question mark and space immediately after the input prompt.
  - If you have a comma separating the input prompt from the variable, BASIC prints the input prompt, skips to the next print zone, and then prints the question mark and space.
- Waits for you to type the value for the variable named in the LINPUT or INPUT LINE statement.

The INPUT LINE statement accepts and stores all characters, including quotation marks, semicolons, and commas, up to and including the line terminator or terminators. LINPUT accepts all characters up to (but not including) the line terminator or terminators. For example:

```
10       INPUT LINE A$
         LINPUT B$
         PRINT A$
         PRINT B$
         PRINT "DONE"
         END

RUN

? "NOW, LOOK HERE!", HE SAID.(RET)
? "NOT THERE, HERE!"(RET)

"NOW, LOOK HERE!", HE SAID.

"NOT THERE, HERE!"
DONE
```

In this example, both INPUT LINE and LINPUT treat your input as a string literal. Therefore, BASIC interprets quotation marks as characters, not as string delimiters.

Note that the INPUT LINE statement includes the line terminator(s) ((RET)) as part of the string literal. This carriage return/line feed pair causes double spacing in the printed output.

INPUT, INPUT LINE, and LINPUT statements can accept data from a terminal or a terminal-format file. See Section 2.3 for I/O to terminal format files.

2.1.2 Providing Input from the Source Program

The following sections describe the READ, DATA, and RESTORE statements. In order to use READ and DATA statements, you must know what data is required when writing the program. These statements do not stop to request data while the program runs. Therefore, your program runs faster than with the INPUT statements.

The RESTORE statement lets you use the same data items more than once.

2.1.2.1 READ Statement

The READ statement reads values from a data block. Its format is:

READ vbl [,vbl]. . .

where:

vbl [,vbl] Is one or more numeric or string variables, either simple or subscripted. Separate all variables with commas.

For example:

```
10 READ A, B%, C$, D(5), E
```

A data pointer keeps track of data read. Each time the READ statement requests data, BASIC retrieves the next available constant from a DATA statement.

You can place a READ statement anywhere in a multi-statement line. However, a READ statement is not valid without at least one DATA statement. If your program contains a READ statement but no DATA statement, BASIC signals the compile-time error "READ without DATA".

2.1.2.2 DATA Statement

The DATA statement contains values that the READ statement supplies to the program. Its format is:

DATA const [,const]. . .

where:

const [,const] Is one or more floating-point, integer, or string constants (quoted or unquoted). Constants must be separated by commas and listed in the same order by data type as the variables requested in the READ statement.

In a DATA statement, integer constants are whole numbers; they cannot be followed by a percent sign. For example:

```
10      ON ERROR GOTO 400
20      READ A%, B%, C%
30      DATA 1%, 2%, 3%
40      PRINT A% + B% + C%
50      GOTO 500
400     PRINT "ERROR NUMBER IS "; ERR
        PRINT "ERROR AT LINE "; ERL
        PRINT "ERROR MESSAGE IS "; ERT$(ERR)
        RESUME 500
500     END

RUNNH


ERROR NUMBER IS 50
ERROR AT LINE 100
ERROR MESSAGE IS %Data format error
```

Because the integer constants in the DATA statement contain percent signs, BASIC signals an error.

You can use an ampersand (&) to continue a DATA statement. For example:

```
10 DATA "ABRAMS", BAKER, CHRISTENSON, &
        DOBSON, "EISENSTADT", FOLEY
```

You can have more than one DATA statement in a program. DATA statements are ignored without at least one READ statement.

Note

BASIC treats all characters between the keyword DATA and the next line number as data. Therefore, a DATA statement must be the last or only statement on a line and the first statement following a DATA statement must be on a numbered line.

Comment fields are not allowed in DATA statements. For example:

```
10 READ A$
20 DATA ABC!COMMENT
```

These statements cause A$ to contain "ABC!COMMENT".

When you compile a program, BASIC creates one data block for each program unit. Each data block is local to the program or subprogram containing it; this means that you cannot share DATA statements between program modules.

The data block contains the values in all DATA statements in that program unit. These values are stored in line number order. Each time BASIC executes a READ statement, it retrieves the next value in the DATA block.

BASIC signals an error if:

- You assign alphabetic characters to a numeric variable. BASIC signals "Data format error" (ERR=50).

- You try to read values into more variables than there are values in DATA statements. BASIC signals "Out of data" (ERR=57).

BASIC ignores excess data in DATA statements.

This example of READ and DATA mixes string and floating-point data types:

```
10 READ TEXT$
20 READ RADIUS
30 DIAMETER = PI * RADIUS * 2
40 DATA "THE DIAMETER IS"
50 DATA 40.5
60 PRINT TEXT$; DIAMETER
70 END

RUN

THE DIAMETER IS 254.469
```

Line 10 reads the first data item in the program: "THE DIAMETER IS". Line 20 reads the second data item: 40.5.

2.1.2.3 RESTORE Statement

The RESTORE statement lets you read the same data more than once. It has no effect without READ and DATA statements. Its format is:

RESTORE

RESTORE resets the data pointer to the beginning of the first DATA statement in the program unit. You can then reread data values. Consider this program:

```
10 READ B,C,D
20 RESTORE
30 READ E,F,G
40 DATA 6,3,4,7,9,2
50 END
```

The READ statement in line 10 reads the first three values in the DATA statement:

B=6
C=3
D=4

The RESTORE statement resets the pointer to the beginning of line 40. During the second READ (line 30), the first three values are read again:

E=6
F=3
G=4

Without the RESTORE statement, line 30 would assign the values:

E=7
F=9
G=2

2.2 Program Output

The PRINT statement displays data on your terminal during program execution. BASIC evaluates expressions before displaying results.

You can also print and format data with the PRINT USING statement (see Chapter 8).

The PRINT statement has the format:

PRINT exp [{ , ; } exp]. . .

where:

exp — Can be any valid expression.

, and ; — Are punctuation marks that determine the cursor position after BASIC prints the expression.

When you use the PRINT statement:

- BASIC precedes positive numbers with a space and negative numbers with a minus sign.
- BASIC prints a space after every number.
- BASIC prints strings with no leading or trailing spaces.

When an element in a list is not a simple variable or constant, BASIC evaluates the expression before printing the value. For example:

```
10      A = 45
        B = 55
20      PRINT A + B
30      END

RUN

 100
```

However, BASIC interprets text inside quotation marks as a string literal. For example:

```
10      A = 45
        B = 55
20      PRINT "A + B"
30      END
```

In this case, the output is:

```
A + B
```

The PRINT statement without an expression prints a blank line. For example:

```
10 PRINT "THIS EXAMPLE LEAVES A BLANK LINE"
20 PRINT
30 PRINT "BETWEEN TWO LINES."
40 END

RUN

THIS EXAMPLE LEAVES A BLANK LINE

BETWEEN TWO LINES.
```

You can place PRINT statements anywhere on a multi-statement line.

2.2.1 Print Zones — The Comma and Semicolon

A terminal line contains zones that are 14 character positions wide. The number of zones in a line depends on the width of your terminal: a 72–character line contains 5 zones, which start in columns 1, 15, 29, 43, and 57. A 132–character line has additional print zones starting at columns 71, 85, 99, and 113.

The PRINT statement formats program output into these zones in different ways, depending on the character that separates the elements to be printed.

If a comma precedes the PRINT item, BASIC prints the item at the beginning of the next print zone. If the last print zone on a line is filled, BASIC continues output at the first print zone on the next line:

```
5       INPUT A ,B ,C ,D ,E ,F
10      PRINT A ,B ,C ,D ,E ,F
20      END

RUN

? 5,10,15,20,25,30(RET)
 5             10            15            20            25
 30
```

BASIC skips one print zone for each extra comma between list elements. For example, this program prints the value of A in the first zone and the value of B in the third:

```
10      A = 5
        B = 10
15 PRINT "FIRST ZONE",,"THIRD ZONE"
20 PRINT A,,B
30 END

RUN

FIRST ZONE                    THIRD ZONE
 5                             10
```

If you separate print elements with a semicolon, BASIC does not move to the next print zone. For example:

```
10       PRINT 10; 20
20       PRINT "ABC"; "XYZ"
30       END

RUNNH


 10  20
ABCXYZ
```

In this example, line 10 prints two numbers. (Printed numbers are preceded by a space or a minus sign and followed by one space.) Line 20 prints two strings.

Placing a comma or semicolon after the last item in a PRINT statement leaves the cursor at the same line:

- A comma specifies that the cursor moves to the next print zone.
- A semicolon specifies that the cursor does not move.

In either case, BASIC waits at that line for another PRINT or INPUT statement.

For example:

```
10       INPUT X,Y,Z
20       PRINT X,Y,
30       PRINT Z
40 END

RUNNH


? 5,10,15
 5             10                  15
```

BASIC prints the current values of X, Y, and Z on one line because a comma follows the last item in line 20.

The following example shows PRINT statements using a comma, a semicolon, and no formatting character after the last print item.

```
10       !No comma after I%, so each element
         !Prints on its own line
         !
         PRINT I% FOR I% = 1% TO 10%
         PRINT
         PRINT
         PRINT
         !
         !A comma follows J%, so each
         !element prints in a separate zone
         !
         PRINT J%, FOR J% = 1% TO 10%
         PRINT
         PRINT
         PRINT
         !
```

(continued on next page)

```
        !A semicolon follows K%, so print
        !elements are packed together
        !
        PRINT K%; FOR K% = 1% TO 10%
        END

RUNNH

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10


 1          2              3              4              5
 6          7              8              9              10


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

Commas and semicolons also let you control the placement of string output:

```
10 PRINT "FIRST ZONE",,"THIRD ZONE",,"FIFTH ZONE"
20 END

RUN

FIRST ZONE                    THIRD ZONE                    FIFTH ZONE
```

The extra comma between strings causes BASIC to skip every other print zone.

2.2.2 Output Format for Numbers and Strings

BASIC prints strings exactly as you type them, with no leading or trailing spaces. It does not print quotation marks unless they are delimited by another matching pair. For example:

```
10 PRINT 'PRINTING "QUOTATION" MARKS'
20 END

RUN

PRINTING "QUOTATION" MARKS
```

BASIC follows these rules for printing numbers:

- When you print numeric fields, BASIC precedes each number with a space or a minus sign and follows it with a space. If a number can be represented exactly by six decimal digits (or fewer) and, optionally, a decimal point, BASIC prints it that way.
- When you print a number whose integer portion is six decimal digits or less (for example, 1234.567), BASIC rounds the number to six digits (1234.57). If the integer portion of a number is seven decimal digits or larger, BASIC rounds the number to six digits and prints it in E format. See Section 1.5.1 for more information about E format.

- When you print a number with magnitude between .1 and 1, BASIC rounds it to six digits. When you print a number having more than six digits, and with magnitude smaller than .1, BASIC rounds it to six digits and prints it in E format.
- 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.
- When you print LONG integers, BASIC prints up to 10 significant digits.

The PRINT statement only displays up to six digits of precision for floating-point numbers. This corresponds to the precision of the SINGLE data type. To display the extra digits in DOUBLE, GFLOAT (VAX–11 BASIC only), and HFLOAT numbers (VAX–11 BASIC only), you must use the PRINT USING statement. See Chapter 8 for more information on PRINT USING.

This example shows how BASIC prints various numbers with single precision:

```
10 FOR I = 1 TO 20
20     PRINT 2^(-I),I,2^I
30 NEXT I
40 END

RUN

 .5                1             2
 .25               2             4
 .125              3             8
 .0625             4             16
 .03125            5             32
 .015625           6             64
 .78125E-02        7             128
 .390625E-02       8             256
 .195313E-02       9             512
 .976563E-03       10            1024
 .488281E-03       11            2048
 .244141E-03       12            4096
 .12207E-03        13            8192
 .610352E-04       14            16384
 .305176E-04       15            32768
 .152588E-04       16            65536
 .767939E-05       17            131072
 .38147E-05        18            262144
 .190735E-05       19            524288
 .953674E-06       20            .104858E+07
```

2.3 Terminal-Format Files

Terminal-format files let you perform simple I/O to disk files. The records in a terminal-format file must be accessed sequentially. That is, you must access the records in the file one by one, from the first to the last. You can add new records only at the end of the file.

Just as the INPUT, LINPUT, and INPUT LINE statements receive information from a terminal, the INPUT #, LINPUT #, and INPUT LINE # statements receive information from a terminal-format file. And, as the PRINT statement sends information to the terminal, the PRINT # statement sends information to a terminal-format file.

Terminal-format files are very useful for creating files to be printed on a line printer (for example, reports) or for supplying a program with moderate amounts of input. However, if you want to use the same file for both input and output, you should not use terminal-format files. Instead, use sequential, relative, or indexed files (see Chapter 9).

Note that you do not have to use a program to create a terminal-format file. You can use a text editor to create a file and insert data, then use a BASIC program to open the file and retrieve the data.

2.3.1 Opening and Closing a Terminal-Format File

You use the OPEN statement to create a file, or to gain access to an existing file. For terminal-format files, the OPEN statement format is:

```
OPEN "file-spec" [FOR INPUT] AS [FILE] #chnl-exp
                 [FOR OUTPUT]
```

where:

file-spec	Is the file specification.
FOR INPUT	Specifies that the file already exists.
FOR OUTPUT	Specifies that a new file is being created.
#chnl-exp	Specifies the channel on which I/O is performed. The channel number must be between 1 and 12 for PDP–11 BASIC–PLUS–2 and between 1 and 99 for VAX–11 BASIC.

If you do not specify either FOR INPUT or FOR OUTPUT, BASIC tries to open an existing file. If the file does not exist, BASIC creates a new one.

The channel specification lets you associate a number with the file for as long as the file is open. All I/O operations to or from the file use this number.

When you are finished accessing a file, you close it with the CLOSE statement. Its format is:

```
CLOSE [#]chnl-exp
```

where:

[#]chnl-exp	Is the channel number of an open file. The pound sign is optional.

2.3.2 Writing Records to a Terminal-Format File

The following example receives information from a terminal, then writes the information to a terminal-format file as a report:

```
100     PRINT "This program creates a daily sales report file named SALES.DAT"
200     OPEN "SALES.DAT" FOR OUTPUT AS FILE #4%
300     PRINT #4%, "Salesperson","Sales Area","Items Sold"
        PRINT
400     INPUT "How many salespersons for today's report"; SALES_PERSONS%
500     FOR I% = 1% TO SALES_PERSONS%
            INPUT "Salesperson's name"; S_NAME$
            INPUT "Sales area"; AREA$
            INPUT "Number of items sold"; ITEMS_SOLD%
            PRINT #4%, S_NAME$, AREA$, ITEMS_SOLD%
        NEXT I%
600     CLOSE #4%
700     END
```

```
RUNNH

This program creates a daily sales report file named SALES.DAT
How many salespersons for today's report? 3
Salesperson's name? JONES
Sales area? NJ
Items sold? 5
Salesperson's name? SMITH
Sales area? NH
Items sold? 6
Salesperson's name? BAINES
Sales area? VT
Items sold? 8

$TYPE SALES.DAT

Salesperson      Sales Area     Items Sold

JONES            NJ              5
SMITH            NH              6
BAINES           VT              8
```

This program first prints a header explaining its purpose, then opens a terminal-format file on channel 4. Line 300 uses the PRINT # statement to place an explanatory header in the file, then prints a blank line.

The program then prompts you for the number of salespersons for which data is to be entered. Line 500 contains a FOR–NEXT loop that prompts for the name, sales area, and items sold for each salesperson. Note that the FOR–NEXT loop executes only as many times as there are salespersons. (See Chapter 3 for more information about FOR–NEXT loops.)

After the data has been entered for each salesperson, the program writes this information to the terminal-format file. Because the response to the first question was three, the FOR–NEXT loop executes three times.

After the last item has been printed to the file, the program closes the file and ends. When you display the file with the $TYPE command, you see that the information is printed under the proper headers. You can also print the file on a line printer. Notice that the PRINT # statement formats the output in print zones as does the PRINT statement.

Chapter 3
Program Control

In a BASIC program, statements are normally executed consecutively, in line number order. Within a program line, statements are executed from left to right. Control statements let you change this sequence of execution by:

- Executing a block of statements repeatedly (looping)
- Executing subroutines
- Suspending or stopping the program
- Executing a block of statements conditionally
- Executing a single statement conditionally
- Executing a single statement repeatedly
- Branching conditionally to another program line or label
- Branching unconditionally to another program line or label

The following sections describe the BASIC statements that let you perform these operations.

3.1 Loops

A program loop is a construct that allows the repeated execution of a set of statements. There are three types of BASIC program loops:

- FOR–NEXT
- WHILE–NEXT
- UNTIL–NEXT

Each type of loop controls the repeated execution of a set of statements. The loop control statements, plus the set of statements in the loop, are called the loop block. You control the number of loop iterations in several ways:

- By specifying starting and ending values for a loop control variable (FOR–NEXT)
- By using a loop construct that performs a logical test (UNTIL–NEXT and WHILE–NEXT)
- By performing a test within the loop block and either explicitly exiting from the loop with the EXIT statement, or explicitly transferring control to the loop control statement (FOR, WHILE, or UNTIL) with the ITERATE statement

Note that the EXIT and ITERATE statements accept label names as arguments. If you intend to use these statements for explicit loop control, you should label your loops.

The following sections describe these loop constructs and when to use them.

3.1.1 FOR–NEXT Loops

FOR–NEXT loops let you specify how many times to repeat a block of statements. If you want to execute a single statement a specified number of times, you should use the FOR modifier. See Section 3.6.3 for more information about the FOR modifier.

In a FOR–NEXT loop, you specify a loop control variable that determines the number of times the loop executes (the number of loop iterations). In a FOR–NEXT loop, the starting and ending values of the loop control variable are known when the loop begins execution. The format of a FOR–NEXT loop is:

```
FOR ctrl-vbl = num-exp1 TO num-exp2 [STEP num-exp3]
.
.
.
NEXT ctrl-vbl
```

where:

ctrl-vbl	Is the control or index variable that is incremented or decremented each time the loop executes. Ctrl-vbl must be a simple (unsubscripted) variable.
num-exp1	Is the starting value for the control variable.
num-exp2	Is the ending value for the control variable.
num-exp3	Is the amount added to the control variable each time the loop executes.

The FOR statement assigns the control variable a starting value, num-exp1, and a terminating value, num-exp2. The optional STEP clause lets you specify the amount, num-exp3, added to the loop control variable after each loop iteration. If you do not specify a STEP clause, the default increment is +1. For example:

```
100     FOR I% = 1% TO 10%
            NEW_ARRAY%(I%) = I%
        NEXT I%
```

This example assigns values to array elements 1 through 10, inclusive.

When the FOR loop block executes, BASIC:

- Evaluates num-exp1 (the starting value) and assigns the result to the control variable.
- Evaluates num-exp2 (the ending value) and num-exp3 (the STEP value) and assigns these results to temporary storage locations.
- Tests to see whether the ending value has been exceeded. If the ending value has already been exceeded, BASIC executes the line following the NEXT statement. If the ending value has not been exceeded, BASIC executes the statements in the loop.
- Adds the STEP value to the control variable and transfers control to the FOR statement, which tests whether the ending value has been exceeded.

Note that BASIC performs the test at the top of the loop. When the control variable exceeds the ending value, BASIC exits the loop, then subtracts the STEP value from the control variable. This means that after loop execution, the value of the control variable is that last used in the loop, not the value that caused loop termination.

Because the starting, ending, and STEP values can be numeric expressions, they are not evaluated until the program runs. Therefore, it is possible to code a FOR–NEXT loop that does not execute. For example:

```
100     COUNTER% = 0%
200     INPUT "START"; START%
        INPUT "FINISH"; FINISH%
        INPUT "STEP VALUE"; STEP_VAL%
300     FOR I% = START% TO FINISH% STEP STEP_VAL%
            COUNTER% = COUNTER% + 1%
        NEXT I%
400     PRINT "This loop executed"; COUNTER%; "times."

RUNNH

START? 0
FINISH? 5
STEP VALUE? -1
This loop executed 0 times.
```

However, if the FOR statement contains integer constants, BASIC can check to see whether the loop will execute when the program is compiled. For example:

```
100     FOR I% = 0% TO 5% STEP -1%
            COUNTER% = COUNTER% + 1%
        NEXT I%
```

When you compile this program, BASIC diagnoses the problem and reports: "Loop will not execute."

Whenever possible, you should use integer variables to control the execution of FOR–NEXT loops. This is because some decimal fractions cannot be represented exactly in a binary computer and the calculation of floating-point control variables is subject to this inherent precision limitation.

The following program uses the DECLARE statement to explicitly declare its variables. See Chapter 5 for more information on DECLARE. The program shows you the different behavior of integer and floating-point control variables:

```
10      DECLARE                                                    &
            INTEGER                                                &
            Integer_loop_index,                                    &
            Loop_count_1,                                          &
            Loop_count_2,                                          &
            REAL                                                   &
            Real_loop_index

        Loop_count_1, Loop_count_2 = 0


        FOR Integer_loop_index = 0 TO 100 STEP 1
            Loop_count_1 = Loop_count_1 + 1
        NEXT Integer_loop_index


        FOR Real_loop_index = 0 TO 10 STEP 0.1
            Loop_count_2 = Loop_count_2 + 1
        NEXT Real_loop_index


        PRINT "Integer  loop count:"; Loop_count_1
        PRINT "Integer  loop end  :"; Integer_loop_index
        PRINT "Real     loop count:"; Loop_count_2
        PRINT "Real     loop end  :"; Real_loop_index
        END

RUNNH

Integer  loop count: 101
Integer  loop end  : 100
Real     loop count: 100
Real     loop end  : 9.9
```

The first loop uses an integer control variable while the second uses a floating-point control variable. The first loop executes 101 times and the second 100 times. After the hundredth iteration of the second loop, the internal representation of the value of Real_loop_index exceeds 10 and BASIC exits the loop. Because the first loop uses integer values to control execution, BASIC does not exit the loop until Integer_loop_index equals 101.

Although it is not recommended programming practice, you can change the value of the control variable inside the loop. For example:

```
10 FOR I% = 1% TO 10% STEP 1%
              I% = I% * 2%
              LOOP_COUNT% = LOOP_COUNT% + 1%
   NEXT I%
   PRINT "I% EQUALS" ;I%
   PRINT "LOOP_COUNT EQUALS"; LOOP_COUNT%
   END

RUNNH

I% EQUALS 14
LOOP_COUNT EQUALS 3
```

BASIC makes only three passes through this loop:

- On the first iteration, I% is assigned a value of one. Line 20 assigns I% a value of two.
- On the second iteration, BASIC increments I% by one (I% now equals three). Line 20 assigns I% a value of six.
- On the third iteration, BASIC increments I% by one (I% now equals seven). Line 20 assigns I% a value of 14, and when BASIC executes the FOR statement a fourth time, the terminating value has already been reached.

Although changing the value of the control variable affects loop execution, changing the STEP value or the terminating value does not. This is because BASIC makes a temporary copy of these values, and these copies are not accessible to you. For example:

```
10      DECLARE                                                 &
            INTEGER                                             &
                Loop_index,                                     &
                Loop_limit,                                     &
                Loop_step

        PRINT "Loop index", "Loop limit", "Loop step"
        Loop_limit = 10
        Loop_step = 1

        FOR Loop_index = 0 TO Loop_limit STEP Loop_step
            PRINT Loop_index, Loop_limit, Loop_step
            Loop_step = 2 * Loop_step
            Loop_limit = Loop_limit - 1
        NEXT Loop_index
        END

RUNNH

Loop index     Loop limit     Loop step
 0              10             1
 1              9              2
 2              8              4
 3              7              8
 4              6              16
 5              5              32
 6              4              64
 7              3              128
 8              2              256
 9              1              512
 10             0              1024
```

Because BASIC compares the control variable to the temporary copy of the terminating value, changing variables Loop_step and Loop_limit has no effect on loop execution.

3.1.2 WHILE–NEXT Loops

WHILE–NEXT loops let you repeatedly execute a block of statements for as long as a given condition is true. If you want to repeat the execution of a single statement for as long as a given condition is true, you should use the WHILE modifier. See Section 3.6.4 for more information about the WHILE modifier.

You can use the keywords WHILE and NEXT to create a multi-line loop. The general format for WHILE loops is:

```
WHILE cond-exp
   statement(s)
NEXT
```

where:

cond-exp Is a conditional expression.

The loop executes repeatedly for as long as the conditional expression is true. For example:

```
5 A = 0
10 WHILE A < 10
20       PRINT A
30       A = A + 1
40 NEXT
50 END
```

BASIC executes lines 20 and 30 as long as variable A is less than 10. Thus the program prints numbers 0 through 9.

A WHILE–NEXT loop, unlike a FOR–NEXT loop, has no explicit control variable. Unless the conditional expression is initially false, your program must either: 1) change a variable in the conditional expression (as in line 30 of the previous example) or 2) transfer control out of the loop. Otherwise, the loop executes indefinitely. For example:

```
100      READ A$
200      WHILE A$ <> "END"
             PRINT A$
             READ A$
500      NEXT
600      DATA ONE,"TWO",THREE, "END"
```

This loop executes until A$ is assigned the value "END" (the final value in the DATA statement).

3.1.3 UNTIL–NEXT Loops

UNTIL–NEXT loops let you repeat the execution of a block of statements until a given condition is true. If you want to repeat the execution of a single statement until a given condition is true, you should use the UNTIL modifier. See Section 3.6.5 for more information about the UNTIL modifier.

You can use the keywords UNTIL and NEXT to create a multi-line loop. The general format for UNTIL–NEXT loops is:

```
UNTIL cond-exp
    statement(s)
NEXT
```

where:

cond-exp Is a conditional expression.

The loop executes repeatedly for as long as the conditional expression is false. For example:

```
10      UNTIL A% = 10%
            PRINT A%
            A% = A% + 1%
        NEXT
50      END
```

The NEXT statement does not increment a control variable: your program must change a variable in the conditional expression or the loop executes indefinitely.

UNTIL and FOR loops differ because BASIC exits UNTIL loops as soon as the test for the terminating condition is met. This test occurs after BASIC executes the NEXT statement and before it executes the UNTIL statement. For example, this loop executes 10 times:

```
10          FOR I% = 1% TO 10%
                A = A + 1
                PRINT A
            NEXT I%
50          END
```

When BASIC leaves the FOR loop, I% equals 10. However, the following UNTIL loop executes only nine times:

```
5           I% = 1%
            UNTIL I% = 10%
                PRINT I%
                I% = I% + 1%
            NEXT
50          END
```

After the ninth iteration, the conditional expression in line 10 is true; control then passes to line 50.

3.1.4 Nested Loops

When a loop block is entirely contained in another loop block, it is called a nested loop. For example:

```
10      DECLARE                                                 &
            INTEGER                                             &
                Column_number,                                  &
                Row_number,                                     &
            REAL                                                &
                Two_dim_array (15%, 15%)
        FOR Row_number = 0% TO 15%
            FOR Column_number = 0% TO 15%
                Two_dim_array (Row_number, Column_number) = 0
            NEXT Column_number
        NEXT Row_number
        END
```

This program declares a two-dimensional array and uses nested FOR–NEXT loops to initialize array elements. The inner loop executes 16 times for each iteration of the outer loop. The program assigns zero to each of the 256 elements in the array.

Note that the inner loop is entirely contained in the outer loop. Nested loops cannot "overlap." For example, this nested loop is invalid:

```
100     DIM X_ARRAY(10,10)
200     FOR I% = 0% TO 10%
300         FOR J% = 0% TO 10%
400             X_ARRAY(I%,J%) = 0
500     NEXT I%
600         NEXT J%
```

When it encounters invalid nested loops, BASIC signals the compile-time error "Illegal loop nesting".

3.1.5 Explicit Loop Control (ITERATE and EXIT Statements)

The ITERATE and EXIT statements let you explicitly control loop execution. These statements can be used only within FOR, WHILE, or UNTIL loops. EXIT can be used only when these loops have been labeled. The format for ITERATE is:

ITERATE [label-name]

where:

label-name Is the label of the first statement of a loop. If you supply a label name, ITERATE transfers control to the specified loop's NEXT statement. If you omit the label name, ITERATE transfers control to the NEXT of the innermost loop that is currently executing.

ITERATE is equivalent to an unconditional branch to the loop's NEXT statement. The format for EXIT is:

EXIT label-name

where:

label-name Is the label of the current loop. Note that unlike ITERATE, EXIT requires that you supply a label name.

EXIT transfers control to the first statement following the specified loop.

This example shows the use of the EXIT statement:

```
10      DECLARE                                                 &
            REAL                                                &
               Operation_result(3),                             &
               Value_1,                                         &
               Value_2

        PRINT "  X      Y      X * Y      X / Y      X + Y      X - Y"
        PRINT

    Process_value:

        WHILE 1% = 1%
            READ Value_1, Value_2

            EXIT Process_value IF Value_2 = 0
```

```
            PRINT USING "##.## ##.## "; Value_1, Value_2;
            Operation_result(0) = Value_1 * Value_2
            Operation_result(1) = Value_1 / Value_2
            Operation_result(2) = Value_1 + Value_2
            Operation_result(3) = Value_1 - Value_2
            FOR I% = 0% TO 3%
                PRINT USING "####.#### "; Operation_result(I%);
            NEXT I%
            PRINT
        NEXT
        DATA                                                        &
         0.50,  0.50,  0.50,  1.00,                                 &
         1.00,  0.50,  1.00,  1.00,                                 &
         0.00,  0.00
32767   END

RUNNH

  X      Y      X * Y      X / Y      X + Y      X - Y

 0.50   0.50     0.2500     1.0000     1.0000     0.0000
 0.50   1.00     0.5000     0.5000     1.5000    -0.5000
 1.00   0.50     0.5000     2.0000     1.5000     0.5000
 1.00   1.00     1.0000     1.0000     2.0000     0.0000
```

The WHILE loop in this program executes until Value_2 is equal to zero. At this point, the EXIT Process_value statement explicitly transfers control out of the loop. Note that the EXIT statement contains an IF modifier to test whether Value_2 is equal to zero.

This example shows the use of both the EXIT and ITERATE statements.

```
100     DECLARE STRING User_string
        Read_loop:
        WHILE 1% = 1%
            LINPUT "Please type a string"; User_string
            IF User_string == ""
                THEN EXIT Read_loop
                ELSE PRINT "Length is ";LEN(User_string)
                     ITERATE Read_loop
            END IF
        NEXT
32767   END
```

This program explicitly exits the loop if you type a carriage return in response to the prompt. If you type a string, the program prints the length of the string and explicitly reexecutes the loop.

3.2 Branching Unconditionally (GOTO Statement)

The GOTO statement specifies the next line that BASIC is to execute, regardless of that line's position in the program. Its format is:

GOTO { lin-num | label-name }

If you specify a line number or label, it must exist in your program.

If the statement at the target line number or label is nonexecutable (for example, a REM statement), BASIC transfers control to the next executable statement.

You can use a GOTO statement to exit from a loop; however, it is better programming practice to use the EXIT statement to do this. You cannot use the GOTO statement to exit from a DEF function definition. If you try to transfer control into or out of a DEF, BASIC signals either "Jump into DEF" or "Jump out of DEF" when the program is compiled. The only valid way to exit from a DEF is with an END DEF, FNEND, EXIT DEF, or FNEXIT statement.

The GOTO statement must be either: 1) the only statement in a block of statements or 2) the last statement in a block of statements. If you place a GOTO statement in the middle of a block, BASIC cannot execute the statements remaining in that block.

Modular programming standards dictate that the number of GOTO statements be held to a minimum. Fewer GOTO statements also make the program easier to read. Use control structures — FOR, UNTIL, and WHILE loops, and the IF–THEN–ELSE and SELECT–CASE constructions — wherever possible.

3.3 Branching Conditionally

Conditional branching is the transfer of program control only when specified conditions are met. Three BASIC statements let you conditionally transfer control to a target statement in your program:

- The ON–GOTO–OTHERWISE statement
- The IF–THEN–ELSE statement
- The SELECT–CASE statement

The ON–GOTO–OTHERWISE statement uses the value of a control variable to determine the target statement. The IF–THEN–ELSE construction uses conditional expressions to determine which program branch to take. The SELECT–CASE allows for the execution of any one of a number of alternative statement blocks depending on the value of the expression you supply.

3.3.1 ON–GOTO–OTHERWISE Statement

The ON GOTO statement transfers control to one of several target statements, depending on the value of a control variable. Its format is:

```
ON int-vbl GOTO target, . . . [OTHERWISE target]
```

where:

target Is a line number or label. Note that a comma cannot appear between the last target and the OTHERWISE keyword.

BASIC tests the value of the control variable, int-vbl. If the value is one, BASIC transfers control to the first target in the list; if the value is two, to the second target, and so on. If the control variable's value is less than one, or greater than the number of targets in the list, BASIC transfers control to the target specified in the OTHERWISE clause. For example:

```
100     PRINT "WOULD YOU LIKE TO CHANGE:"
        PRINT "1. FIRST NAME"
        PRINT "2. LAST NAME"
200     INPUT CHOICE%
300     ON CHOICE% GOTO 1000, 1100 OTHERWISE 32766
1000    INPUT "FIRST NAME"; FIRSTNAME$
1050    GOTO 32767
```

```
1100     INPUT "LAST NAME"; LASTNAME$
         .
         .
         .
32766    PRINT "INVALID CHOICE"
32767    END
```

This program prompts for changes to one of two variables. If you enter "1", BASIC transfers control to line 1000; if you enter "2", BASIC transfers control to line 1100. If you enter any other value, BASIC transfers control to line 32766.

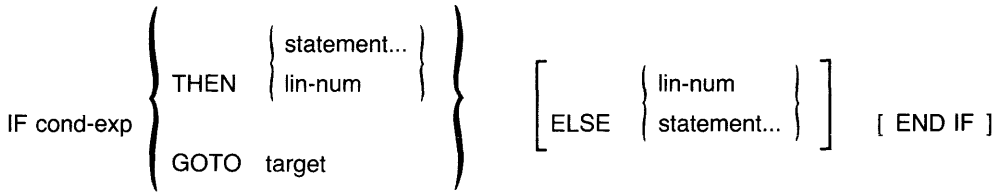
If you do not supply an OTHERWISE clause and the control variable's value is less than one or greater than the number of targets, BASIC signals "ON statement out of range" (ERR = 58).

This example shows the use of labels in ON–GOTO–OTHERWISE:

```
100     Ask:
          PRINT "Would you like to:"
          PRINT "Change first name (1)"
          PRINT "Change last name (2)"
          PRINT "Change middle initial (3)"
          PRINT "Exit from the program (4)"
          INPUT "Type a number"; N%
          ON N% GOTO First, Last, Middle, Done OTHERWISE Bad_input

     First:
          PRINT
          INPUT "First name";First_name$
          !       .
          !       .
          !       .
          PRINT
          GOTO Done

     Last:
          PRINT
          INPUT "Last name";Last_name$
          !       .
          !       .
          !       .
          PRINT
          GOTO Done

     Middle:
          PRINT
          INPUT "Middle initial";Middle_I$
          !       .
          !       .
          !       .
          PRINT
          GOTO Done

     Bad_input:
          PRINT
          PRINT "Invalid value, please try again"
          PRINT
          GOTO Ask

     Done:
          END
```

3.3.2 IF–THEN–ELSE and END IF Statements

The IF–THEN–ELSE statement evaluates a conditional expression and uses the resulting value to determine which block of statements to execute next. Its format is:



where:

target Is either a line number or a label.

The END IF keywords terminate the IF statement. All statements between the ELSE keyword and the next line number or END IF are part of the ELSE clause.

Note that IF–THEN–ELSE is most useful for executing blocks of statements. If you want to conditionally execute a single statement, you should use the IF modifier. See Section 3.6.1 for more information on the IF modifier.

In the IF–THEN–ELSE statement, BASIC first evaluates the conditional expression. If it is true, BASIC executes the statements in the THEN clause and skips the statements in the ELSE clause. If the conditional expression is false, BASIC skips the statements in the THEN clause and executes the statements in the ELSE clause. For example:

```
10 REM THIS PROGRAM FINDS A NUMBER'S SQUARE ROOT
20 INPUT "INPUT NUMBER"; NUMBER
30       IF (NUMBER < 0)
         THEN
             NUMBER = NUMBER * (-1)
             PRINT "THAT SQUARE ROOT IS IMAGINARY"
             PRINT "THE SQUARE ROOT OF ITS ABSOLUTE VALUE IS";
             PRINT SQR(NUMBER)
         ELSE
             PRINT "THE SQUARE ROOT IS"; SQR(NUMBER)
         END IF
40 END

RUNNH

INPUT NUMBER? -9
THAT SQUARE ROOT IS IMAGINARY
THE SQUARE ROOT OF ITS ABSOLUTE VALUE IS 3
```

BASIC evaluates the conditional expression "NUMBER < 0" in line 30. If the input value of variable NUMBER is less than zero:

- The conditional expression is true.

- BASIC executes the four statements in the THEN clause and skips the statement in the ELSE clause. BASIC transfers control to the statement following the END IF, or, if there is no END IF statement, to the next line number.

If the value of NUMBER is greater than or equal to zero:

- The conditional expression is false.

- BASIC skips the statements in the THEN clause and executes the statement in the ELSE clause.

The THEN and ELSE clauses can contain line numbers instead of statements. For example:

```
100 INPUT "TYPE A VALUE FOR VARIABLE 'A'"; A
110 IF A = 0 THEN 120 ELSE 140
120 PRINT "A EQUALS ZERO"
130 GOTO 150
140 PRINT "A DOES NOT EQUAL ZERO"
150 END
```

However, recommended programming practice is to include the GOTO keyword for readability and ease of program maintenance.

If you do not specify an ELSE clause and the conditional expression is false, BASIC executes the first numbered line following the IF statement. For example:

```
10 INPUT "TYPE A VALUE FOR VARIABLE 'A'";A
20 IF A=0 THEN PRINT "A EQUALS ZERO"\GOTO 40
30 PRINT "A DOES NOT EQUAL ZERO"
40 END

RUNNH

TYPE A VALUE FOR VARIABLE 'A'? 5
A DOES NOT EQUAL ZERO
```

The END IF statement lets you explicitly end an IF–THEN–ELSE block. Its format is:

```
END IF
```

The END IF statement is useful because it is not always easy to tell where the IF–THEN–ELSE block ends. For example:

```
100     IF (A% => 50%)
            THEN PRINT "A% is greater than or equal to 50"
            ELSE PRINT "A% is less than 50"
        GOTO 1000
```

In this example, the GOTO statement is part of the ELSE clause. However, the indentation makes it look as though the GOTO should be executed whether the conditional expression is true or false. With the END IF statement, you can explicitly end the IF–THEN–ELSE:

```
100     IF (A% => 50%)
        THEN PRINT "A% is greater than or equal to 50"
        ELSE PRINT "A% is less than 50"
        END IF
        GOTO 1000
```

In this example, the END IF statement explicitly ends the IF block; therefore, BASIC does not treat the GOTO statement as part of the ELSE clause. If there is no END IF statement, the next numbered line marks the end of the IF–THEN–ELSE block. Note that while line numbers terminate IF–THEN–ELSE blocks, labels do not.

If an IF–THEN–ELSE block is preceded by a label, you can use the EXIT statement to transfer control out of the IF–THEN–ELSE block. For example:

```
100     DECLARE LONG CONDITION
200     ASK_BLOCK:
        !
        PRINT "Please type a TRUE/FALSE value"
        INPUT "Zero for FALSE, minus one for TRUE"; CONDITION
        !
        IF_BLOCK:
        !
        IF CONDITION = 0%
        THEN
                PRINT "False"
                GOTO DONE
        ELSE
                PRINT "True"
                EXIT IF_BLOCK IF CONDITION = -1%
                PRINT "But value is not equal to minus"
                PRINT "one.  Please try again"
                GOTO ASK_BLOCK
        END IF
        !
        DONE:   END
```

In this program, if the CONDITION variable is anything other than zero, control is transferred to the ELSE clause and the first statement in the ELSE clause is always executed. The second statement in the ELSE clause transfers control out of the IF block if the CONDITION variable is minus one. The remaining statements in the ELSE clause are executed only if the value of CONDITION is other than minus one.

The EXIT statement transfers control to the statement immediately following the END IF statement. If there is no END IF statement, EXIT transfers control to the next line-numbered statement.

3.3.3 SELECT–CASE Statement

The SELECT–CASE construction lets you specify an expression, the possible values the expression may have, and a list of statements to be executed for each possible value. The END SELECT statement terminates the SELECT–CASE construction. The code between SELECT and END SELECT is called a

SELECT block, and the code between CASE statements is called a CASE block. Each statement in a SELECT block can have its own line number. The SELECT–CASE format is:

```
SELECT select-exp
        CASE [relation] case-item
            statement(s)
        CASE case-item [,...]
            statement(s)
        CASE case-item TO case-item [,case-item TO case-item][,...]
            statement(s)
        [CASE ELSE
            statement(s)]
END SELECT
```

where:

select-exp Is the expression to be tested. Select-exp can be numeric or string.

relation Is a relational operator specifying the test to be performed.

case-item Is either an expression to be compared with the select expression or a range of values separated with the keyword TO. Multiple ranges are valid and can be separated with commas.

ELSE Specifies the statements to be executed if no match is found in the previous CASE statements.

When a match is found between the select-expression and a case-item, all the statements in the following CASE block are executed. Control is then transferred to the statement following the END SELECT statement.

You can specify case-items whose values overlap. If there are overlapping values, the statements associated with the first matching CASE statement are executed. For example:

```
1000    SELECT X% + Y% + Z%
        CASE = 0%
            PRINT "X% + Y% + Z% equals 0"
        CASE 1% TO 100%
            PRINT "X% + Y% + Z% is between 1 and 100"
        CASE 90% TO 200%
            PRINT "X% + Y% + Z% is between 101 and 200"
        CASE ELSE
            PRINT "X% + Y% + Z% is not in selected range"
        END SELECT
```

In this example, values between 90 and 100 are always handled by the first case block.

Note that if you do not supply a CASE ELSE and no matches are found, control is transferred to the statement following the END SELECT statement.

Specifying a list of CASE items is also very useful, for example, when coding an error handler:

```
1       ON ERROR GOTO Err_routine
        !.
        !.
        !.
```

(continued on next page)

```
100
    Get_file_spec:
        !.
        !.
        !.
200
    Input_number:
        !.
        !.
        !.
18000   GOTO 32767


    Err_routine:

        SELECT ERR

        CASE 1,2,5,6

                !
                ! 1 = ?Bad directory for device
                ! 2 = ?Illegal file name
                ! 5 = ?Can't find file or account
                ! 6 = ?Not a valid device
                !
                PRINT "Problem with file specification"
                RESUME 100

        CASE 48,51,52

                !
                ! 48 = ?Floating point error or overflow
                ! 51 = ?Integer error or overflow
                ! 52 = ?Illegal number
                !
                PRINT "Bad number"
                RESUME 200

        CASE ELSE

                PRINT "Unexpected error -- exiting"
                ON ERROR GOTO 0

        END SELECT

32767   END
```

The first CASE block traps errors that might be encountered when trying to open a file. The second CASE block traps errors that might be encountered when assigning numeric values from the terminal. The CASE ELSE block tells BASIC to handle any other errors.

If a SELECT block is preceded by a label, you can use the EXIT statement to transfer control out of the SELECT block. For example:

```
100     DECLARE LONG S_NUM
200     INPUT "Please type a value between 1 and 9"; S_NUM
300     !
        SELECT_BLOCK:
        SELECT S_NUM
        CASE 1 TO 3
                PRINT "Value is 1, 2, or 3"
                EXIT SELECT_BLOCK IF S_NUM = 1
                PRINT "Value is 2 or 3"
                EXIT SELECT_BLOCK IF S_NUM = 2
                PRINT "Value is 3"
```

```
        CASE 4 TO 6
                PRINT "Value is 4, 5, or 6"
                EXIT SELECT_BLOCK IF S_NUM = 4
                PRINT "Value is 5 or 6"
                EXIT SELECT_BLOCK IF S_NUM = 5
                PRINT "Value is 6"
        CASE 7 TO 9
                PRINT "Value is 7, 8, or 9"
                EXIT SELECT_BLOCK IF S_NUM = 7
                PRINT "Value is 8 or 9"
                EXIT SELECT_BLOCK IF S_NUM = 8
                PRINT "Value is 9"
        CASE ELSE
                PRINT "Value out of range"
        END SELECT
400     GOTO 200
32767   END
```

This program transfers control out of the labeled SELECT block if S_NUM contains certain values.

The EXIT statement transfers control to the statement immediately following the END SELECT statement.

3.4 Executing Local Subroutines

In BASIC, a subroutine is a block of code accessed by a GOSUB or ON GOSUB statement. It is always in the same program unit as the statement that calls it. The RETURN statement in the subroutine returns control to the statement immediately following the GOSUB.

The first line of a subroutine can be any valid BASIC statement, including a REM statement. You need not transfer control to the first line of the subroutine. Instead, you can include several entry points into the same subroutine. Similarly, you can nest subroutines by using a GOSUB or ON GOSUB statement in a subroutine.

Variables and data in a subroutine are global to the program unit in which the subroutine resides.

3.4.1 GOSUB and RETURN Statements

The GOSUB statement unconditionally transfers control to a line in a subroutine. The last statement in a subroutine is a RETURN statement, which returns control to the first statement after the calling GOSUB. A subroutine can contain more than one RETURN statement so you can return control to the calling if a specified condition is true. The GOSUB statement's format is:

```
GOSUB target
```

where:

target Is a line number or label entry point to the subroutine. The line number or label must exist in the current program.

The RETURN statement's format is:

```
RETURN
```

For example:

```
100     A=5
        GOSUB Two_times
        PRINT A
```

(continued on next page)

```
        A=15
        GOSUB Two_times
        PRINT A

        A=25
        GOSUB Two_times
        PRINT A

        GOTO Done

    Two_times:

        ! This is the subroutine entry point
        A=A*2
        RETURN

    Done:

        END

RUNNH

 10
 30
 50
```

The program first assigns a value of five to the variable A, then transfers control to the subroutine labeled Two_times. The subroutine replaces the value of A with A multiplied by two. The subroutine's RETURN statement transfers control to the first PRINT statement, which displays the changed value. The program calls the subroutine twice more, with different values for A. Each time, the RETURN transfers control to the statement immediately following the corresponding GOSUB.

Note that BASIC signals "RETURN without GOSUB" if it encounters a RETURN statement without having encountered a previous GOSUB or ON GOSUB statement.

3.4.2 ON–GOSUB–OTHERWISE Statement

The ON–GOSUB–OTHERWISE statement transfers control to one of several target subroutines depending on the value of a numeric expression. A RETURN statement returns control to the first statement after the calling ON GOSUB. A subroutine can contain more than one RETURN statement so you can return control to the calling statement if a specified condition is true. The ON–GOSUB–OTHERWISE statement's format is:

ON int-exp GOSUB target, target. . . [OTHERWISE target]

BASIC tests the value of the integer expression. If the value is one, control transfers to the first line number or label in the list; if two, to the second line number or label, and so on. If the control variable's value is less than one or greater than the number of targets in the list, BASIC transfers control to the line number or label specified in the OTHERWISE clause. If you do not supply an OTHERWISE clause and the control variable's value is less than one or greater than the number of targets, BASIC signals "ON statement out of range" (ERR = 58). For example:

```
100     INPUT "Please enter 1, 2 or 3"; A%
200     ON A% GOSUB 1000, 2000, 3000, OTHERWISE ERR_ROUTINE
        GOTO Done
1000    PRINT "That was a 1"
1100    RETURN
```

```
2000     PRINT "That was a 2"
2100     RETURN
3000     PRINT "That was a 3"
3100     RETURN
    ERR_ROUTINE:
         PRINT "Out of range"
         GOTO 100
    Done:
         END
```

If a GOSUB or ON GOSUB statement occurs in a DEF, its target statement must also be in that DEF. This means that you cannot share a subroutine among DEFs. However, you can code this as a DEF with no arguments instead of as a subroutine. This DEF can be invoked by other DEFs.

3.5 Suspending and Halting Program Execution

Two statements suspend program execution:

- SLEEP
- WAIT

A BASIC program automatically halts and closes all files after execution of its last statement. However, you can explicitly halt program execution by using:

- The STOP statement
- The END statement

The STOP statement does not close files. It can appear anywhere in a program. The END statement closes files and must be the last statement in a main program.

3.5.1 SLEEP Statement

The SLEEP statement suspends program execution for a specified number of seconds. Its format is:

SLEEP int-exp

where:

int-exp Is the number of seconds BASIC waits before resuming execution.

For example:

```
10 INPUT "TYPE A STRING OF CHARACTERS";C$
20 SLEEP 120%
30 PRINT C$
40 END
```

This program waits two minutes after receiving the input string, then prints it.

The SLEEP statement is useful if you have a program that depends on another program for data. Instead of constantly checking for a condition, the SLEEP statement lets you check at specified intervals.

3.5.2 WAIT Statement

You use the WAIT statement only with terminal input statements such as INPUT, INPUT LINE, and LINPUT. Its format is:

WAIT int-exp

where:

int-exp Specifies the number of seconds BASIC waits for response to an input statement. This number must be between zero and 255, inclusive.

For example:

```
10 WAIT 30%
20 INPUT "YOU HAVE THIRTY SECONDS TO TYPE YOUR PASSWORD"; PSW$
30 END
```

This program prompts for input, then waits 30 seconds for your response. If the program does not receive input in the specified time, BASIC signals "Keyboard wait exhausted" (ERR = 15).

The WAIT statement affects all subsequent INPUT, INPUT LINE, LINPUT, MAT INPUT, and MAT LINPUT statements. To disable a previously specified WAIT, use WAIT 0%. For example:

```
10 WAIT 30%
20 INPUT "YOU HAVE 30 SECONDS TO TYPE YOUR PASSWORD"; PSW$
30 WAIT 0%
40 PRINT "DO YOU WANT TO USE:"
50 PRINT "ORDER ENTRY SYSTEM?"
60 PRINT "ACCOUNTS RECEIVABLE?"
70 PRINT "OR ACCOUNTS PAYABLE?"
80 INPUT APPLICATION_PACKAGE$
```

The INPUT statement in line 20 requires a response within 30 seconds. The WAIT 0% statement in line 30 disables this 30–second requirement for subsequent INPUT statements.

3.5.3 STOP Statement

The STOP statement is a debugging tool that lets you check the flow of program logic. STOP suspends program execution but does not close files. Its format is:

STOP

When BASIC executes a STOP statement, it prints "STOP at line lin-num". If the program executes in the BASIC environment, BASIC then prompts with the command level prompt. In response, you can type:

- Immediate mode statements
- CONT (to continue program execution)

You use the STOP statement when debugging in immediate mode in the BASIC environment. Refer to Chapter 1 in *BASIC on VAX/VMS Systems*, *BASIC on RSX–11M/M–PLUS Systems*, or *BASIC on RSTS/E Systems* for more information on immediate mode.

If a program containing a STOP statement is compiled, linked or task-built, and executed, you see a pound sign (#) prompt when the STOP statement is encountered. In response to the pound sign prompt, you can type:

- CONT (to continue program execution)
- EXIT (to return to system command level)

The features of the BASIC environment are not available if you execute the program outside of the BASIC environment.

3.5.4 END Statement

The END statement marks the end of a main program. Execution of the END statement closes all files and halts execution. Its format is:

```
END
```

The END statement is optional. However, you should include it for good programming practice. The END statement must be the last statement in the main program.

If you run your program in the BASIC environment, the END statement returns your terminal to BASIC command level. If you execute the program outside of the BASIC environment, the END statement returns your terminal to system command level.

3.6 Executing Statements Conditionally

Statement modifiers are keywords that qualify or restrict a statement. They let you: 1) execute a statement conditionally or 2) create an implied loop.

BASIC has five statement modifiers:

- IF
- UNLESS
- FOR
- UNTIL
- WHILE

A statement modifier affects only the statement immediately preceding it. You can modify only executable statements. Declarative statements are not modifiable. For example, these statements are not modifiable:

- COMMON
- DECLARE
- EXTERNAL
- FUNCTION
- MAP
- SUB

3.6.1 IF Modifier

The IF modifier tests a conditional expression. If the conditional expression is true, BASIC executes the statement. If false, BASIC executes the next statement. The format is:

```
statement IF cond-exp
```

where:

cond-exp Is any conditional expression.

For example:

```
10 PRINT X IF X <> 0
```

This takes less space than the equivalent IF–THEN–ELSE statement:

```
10      IF X <> 0
        THEN
            PRINT X
        END IF
```

No THEN or ELSE clause can follow an IF modifier. However, a statement in a THEN or ELSE clause can be followed by a modifier. In this case, IF applies only to the preceding statement. For example:

```
10      IF A = B
        THEN
            PRINT A IF A = 3
        ELSE
            PRINT B IF B > 0
        END IF
```

3.6.2 UNLESS Modifier

The UNLESS modifier tests a conditional expression. BASIC executes the modified statement only if the conditional expression is false. Its format is:

```
statement UNLESS cond-exp
```

where:

cond-exp Can be any conditional expression.

For example:

```
10 PRINT A UNLESS A = 0
```

This is equivalent to:

```
10 PRINT A IF A <> 0
```

3.6.3 FOR Modifier

The FOR modifier creates a loop on a single line. Its format is:

```
statement FOR ctrl-vbl = num-exp1 TO num-exp2 [STEP num-exp3]
```

where:

ctrl-vbl — Is the control or index variable that is incremented or decremented each time the loop executes. Ctrl-vbl must be a simple (unsubscripted) variable.

num-exp1 — Is the starting value for the control variable.

num-exp2 — Is the ending value for the control variable.

num-exp3 — Is the amount added to the control variable each time the loop executes.

For example:

```
10 A = A + 1 FOR I% = 1% TO 10%
```

This single-line loop is equivalent to:

```
10 FOR I% = 1% TO 10%
20       A = A + 1
30 NEXT I%
```

3.6.4 WHILE Modifier

The WHILE modifier repeats a statement as long as a conditional expression is true. Like the UNTIL and FOR modifiers, it lets you create single line loops. Its format is:

statement WHILE cond-exp

where:

cond-exp — Can be any conditional expression.

For example:

```
10 A = A / 2 WHILE ABS(A) > .1
```

BASIC replaces the value of A with A/2 as long as the absolute value of A is greater than 1/10.

The WHILE modifier may create an infinite loop if its parts are logically unrelated. For example:

```
10 A = 1
20 X = X + 1 WHILE A < 1000
```

The statement in line 20 never reaches a terminating condition.

3.6.5 UNTIL Modifier

The UNTIL modifier, like the FOR modifier, creates a single-line loop. However, instead of using a formal loop variable, you specify the terminating condition with a conditional expression. Its format is:

statement UNTIL cond-exp

where:

cond-exp — Can be any conditional expression.

The modified statement executes repeatedly as long as the condition is false. For example:

```
10 B = B + 1 UNTIL (A - B) < .0001
```

Because of inherent precision limitations, you should not use UNTIL loops of the form:

```
10 Z = Z + 1 UNTIL Z/5 = 100
```

Because Z/5 may never exactly equal 100, this loop could execute indefinitely.

3.6.6 Nested Modifiers

You can append more than one modifier to a statement. This is called "nesting" modifiers. BASIC evaluates nested modifiers from right to left. If the test of the rightmost modifier fails, control passes to the next statement, not to the preceding modifier on the same line. For example:

```
10              A = 5
                B = 10
                C = 15
20 PRINT A IF A = 5 UNLESS C = 15
30 PRINT B UNLESS C = 15 IF B = 10
40 PRINT C IF B = 10 UNLESS C = 5
50 END

RUNNH

 15
```

In this example:

- Line 10 assigns values to variables A, B, and C.
- BASIC tests the rightmost modifier in line 20 (UNLESS C=15). Because C equals 15, the test fails. BASIC passes control to line 30 without testing the IF modifier in line 20.
- BASIC tests the rightmost modifier in line 30 (IF B=10). Because B equals 10, this condition is met. BASIC then tests the UNLESS modifier in line 30. Because C equals 15, this test fails, and control passes to line 40.
- BASIC tests the rightmost modifier in line 40 (UNLESS C=5). Because C does not equal 5, this condition is met. BASIC then tests the IF modifier in line 40. Because B equals 10, this condition is also met. BASIC prints the variable C.
- BASIC passes control to line 50, and the program ends.

Chapter 4
Strings

This chapter defines fixed-length and dynamic strings, explains which you should choose for your application, and shows you how to use them.

A string is a sequence of ASCII characters. BASIC allows three types of strings:

- Dynamic strings
- Fixed-length strings
- Virtual array strings

A string whose length never changes is a fixed-length, or static, string. A string whose length can change during program execution is a dynamic string. Strings in virtual arrays have both fixed-length and dynamic attributes. That is, string virtual arrays have a specified maximum length between zero and 512 characters. During program execution, the length of an element in a string virtual array can change; however, the length is always between zero and the maximum string size specified when the array was created. See Section 4.8 and Chapter 9 for more information about virtual arrays.

String constants are always fixed-length. String variables can be either fixed-length or dynamic. A string variable is fixed-length if it is named in a COMMON, MAP, or RECORD (VAX–11 BASIC only) statement. If a string variable is not part of a MAP, COMMON, RECORD, or virtual array, it is a dynamic string.

If a string variable is fixed-length, its length does not change, regardless of the statement you use to modify it. The length of a dynamic string variable may or may not change, depending on the statement used to modify it. See Table 4–1.

Table 4–1: String Modification

Statement	For Fixed-Length Strings, Changes	For Dynamic Strings, Changes
LET	Value	Value and Length
LSET	Value	Value
RSET	Value	Value
Terminal I/O Statements*	Value	Value and Length

* Terminal I/O statements include INPUT, INPUT LINE, LNPUT, MAT INPUT, and so on.

It is more efficient to change a fixed-length string than a dynamic string. Creating or modifying a dynamic string often causes BASIC to create new storage, and this increases processor overhead. Modifying fixed-length strings involves less overhead because BASIC simply reuses existing storage.

Although dynamic strings are less efficient, they are often more flexible than fixed-length strings. For example, to concatenate strings, you just use the LET statement to assign the concatenated value to a dynamic string variable. You do not have to worry about BASIC truncating the string or adding trailing spaces to it. However, if this destination variable is fixed-length, you must make sure that it is long enough to receive the concatenated string, or BASIC truncates the new value to fit the destination string. Similarly, if you use LSET or RSET to concatenate strings, you must ensure that the destination variable is long enough to receive the data.

4.1 Using Dynamic Strings

The LET, LSET, and RSET statements all operate on dynamic strings as well as fixed-length strings. However, LSET and RSET do not change the length of a dynamic string. For example:

```
10        LET A$ = "ABC"
          LET B$ = "XYZ"
          LET C$ = "      "
20        LSET A$ = A$ + B$
30        LSET C$ = A$ + B$
40        PRINT A$
40        PRINT C$
60        END

RUNNH

ABC
ABCXYZ
```

Line 10 assigns the value ABC to A$, XYZ to B$, and six spaces to C$. At line 20, LSET assigns A$ the value of A$ concatenated with B$. Because the LSET statement does not change the length of the destination string variable, only the first three characters of the expression A$ + B$ are assigned to A$. Line 30 uses LSET to assign C$ the value of A$ concatenated with B$. Because C$ already has a length of six, this statement assigns the value ABCXYZ to it.

Like the LET statement, the INPUT, INPUT LINE, and LINPUT statements can change the length of a dynamic string, but they cannot change the length of a fixed-length string. For example:

```
10      LET A$ = ""
20      PRINT LEN(A$)
30      INPUT A$
40      PRINT A$
50      PRINT LEN(A$)
60      END

RUNNH

 0
? THIS IS A TEST
THIS IS A TEST
 14
```

Line 10 assigns the null string to variable A$. Line 20 uses the LEN function to show that the null string has a length of zero. Line 30 uses the INPUT statement to assign a new value to A$, and lines 40 and 50 print the new value and its length.

You should not confuse the null string with a null character. A null character is one whose ASCII numeric code is zero. The null string is a string whose length is zero.

4.2 Using Fixed-Length Strings

If a string variable is part of a MAP, COMMON, or virtual array, a LET, INPUT, LINPUT, or INPUT LINE statement changes its value, but not its length. For example:

```
10      MAP (FIELDS) STRING FULL_NAME = 10,                    &
                     ADDRESS = 10,                             &
                     CITY_STATE = 10,                          &
                     ZIP = 10
20      LINPUT "NAME"; FULL_NAME
        LINPUT "ADDRESS"; ADDRESS
        LINPUT "CITY AND STATE" CITY_STATE
        LINPUT "ZIP CODE"; ZIP
30      EMPLOYEE_RECORD$ = FULL_NAME + HT + ADDRESS + HT &
            + CITY_STATE + HT + ZIP
40      PRINT EMPLOYEE_RECORD$
50      END

RUNNH

NAME? JOE SMITH
ADDRESS? 66 GRANT AVENUE
CITY AND STATE? NEW YORK NY
ZIP? 01001

JOE SMITH            66 GRANT A    NEW YORK N 01001
```

The MAP statement at line 10 explicitly assigns a length to each string variable. Because the LINPUT statements cannot change this length, BASIC truncates values to fit the ADDRESS and CITY_STATE variables. Because the ZIP variable is longer than the assigned value, BASIC left-justifies the assigned value and pads it with spaces. Line 30 uses the compile-time constant HT (horizontal tab) to separate fields in the employee record.

4.3 Assigning and Justifying String Data

The LSET and RSET statements, unlike LET, do not change a dynamic string variable's length. LSET left-justifies string data, that is, the value you assign with LSET is always in the leftmost position(s) of the destination variable. Similarly, RSET right-justifies string data; the value you assign with RSET is always in the rightmost position(s) of the destination variable.

If the value you assign is longer than the destination variable, LSET and RSET truncate the value to the size of the destination variable. LSET truncates on the right, and RSET truncates on the left.

Similarly, if the value you assign is shorter than the destination variable, LSET and RSET *pad* the value to match the size of the destination variable. Padding consists of the spaces required to make the length of the value match the length of the destination variable. LSET pads on the right, and RSET pads on the left.

4.4 LET Statement

The LET statement assigns string data to a string variable. The keyword LET is optional. Its format is:

[LET] str-vbl = str-exp

where:

str-vbl Is a string variable.

str-exp Is a string expression.

The LET statement changes the length of dynamic strings but does not change the length of fixed-length strings. For example:

```
10      MAP (TEST) STRING ABC = 10
        DECLARE STRING XYZ
20      ABC = "ABC"
        XYZ = "XYZ"
        PRINT ABC, LEN(ABC)
        PRINT XYZ, LEN(XYZ)
        ABC = "A"
        XYZ = "X"
        PRINT ABC, LEN(ABC)
        PRINT XYZ, LEN(XYZ)

RUNNH

ABC             10
XYZ             3
A               10
X               1
```

This program first creates a fixed-length string named ABC by declaring the string in a MAP statement. The program then creates a dynamic string named XYZ by declaring it in a DECLARE statement. Line 20 assigns a 3–character value to both variable ABC and XYZ, then prints the value and the length of the string variables. Variable ABC continues to have a length of 10: the three characters assigned, plus seven spaces for padding. The length of the dynamic variable changes with the values assigned to it.

4.4.1 LSET Statement

The LSET statement left-justifies data and assigns it to a string variable, without changing the variable's length. Its format is:

LSET str-vbl = str-exp

where:

str-vbl Is a string variable.

str-exp Is a string expression.

If the string expression's value is shorter than the string variable's current length, LSET left-justifies the expression and pads the string variable with spaces. For example:

```
10      LET TEST$ = "ABCDE"
20      LSET TEST$ = "XYZ"
30      PRINT "'"; TEST$; "'"
40      END

RUNNH

'XYZ  '
```

The LET statement in line 10 creates the 5–character string variable TEST$. The LSET statement in line 20 assigns the string XYZ to the variable TEST$ but does not change the length of TEST$. Because TEST$ has a length of five, the LSET statement pads the string XYZ with two spaces when assigning the value. The PRINT statement in line 30 shows that TEST$ includes these two spaces.

LSET left-justifies a string expression longer than the string variable and truncates it on the right. For example:

```
10      LET TEST$ = "ABCDE"
20      LSET TEST$ = "12345678"
30      PRINT TEST$
40      END

RUNNH

12345
```

The LET statement in line 10 creates the 5–character string variable TEST$. The LSET statement in line 20 assigns the characters "12345" to TEST$. Because LSET does not change the string variable's length, it truncates the last three characters (678).

4.4.2 RSET

The RSET statement right-justifies data and assigns it to a string variable without changing the variable's length. Its format is:

RSET str-vbl = str-exp

where:

str-vbl Is a string variable.

str-exp Is any string expression.

RSET right-justifies a string expression shorter than the string variable and pads it with spaces on the left. For example:

```
10      LET TEST$ = "ABCDE"
20      RSET TEST$ = "XYZ"
30      PRINT "X" + TEST$
40      END

RUNNH

X   XYZ
```

The LET statement in line 10 creates the 5–character string variable TEST$. The RSET statement in line 20 assigns the string XYZ to TEST$ but does not change the length of TEST$. Because TEST$ is five characters long, the RSET statement pads XYZ with two spaces when assigning the value. The PRINT statement in line 30 shows that the concatenation of "X" and TEST$ includes these two spaces.

If the string expression's value is longer than the string variable, RSET right-justifies the string expression and truncates characters on the left to fit the string variable. For example:

```
10      LET TEST$ = "ABCDE"
20      RSET TEST$ = "987654321"
30      PRINT TEST$
40      END

RUNNH

54321
```

The LET statement in line 10 creates a 5–character string variable, TEST$. The RSET statement assigns "54321" to TEST$. RSET, which does not change the variable's length, truncates "9876" from the left side of the string expression.

Note that, when using LSET and RSET, padding can become part of the data:

```
100     LET A$ = '12345'
        LSET A$ = 'ABC'
        LET B$ = '12345678'
        RSET B$ = A$
        PRINT B$

RUNNH
   ABC
```

In this example, the final value of B$ is " ABC ".

4.5 Manipulating String Data with String Functions

When used with the LET statement, BASIC string functions let you manipulate and modify strings. These functions let you:

- Determine the length of a string (LEN)
- Search for the position of a set of characters in a string (POS, INSTR)
- Extract segments from a string (SEG$, MID$, LEFT$, and RIGHT$)

- Create a string of any length, made up of any single character (STRING$)
- Create a string of spaces (SPACE$)
- Remove trailing spaces and tabs from a string (TRM$)
- Edit a string (EDIT$)

4.5.1 Finding a String's Length (LEN Function)

The LEN function returns the number of characters in a string as an integer value. Its format is:

LEN(str-exp)

where:

str-exp Is any string expression.

The length of the string expression includes leading and trailing blanks. For example:

```
10      ALPHA$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20      PRINT LEN(ALPHA$)
30      Z$ = "ABC" + "   " + "XYZ"
40      PRINT LEN(Z$)
50      END

RUNNH

 26
 8
```

The variable Z$ is set equal to "ABC XYZ", which has a length of eight.

4.5.2 Position Functions (POS and INSTR)

POS and INSTR both search a string for a group of characters (a substring). Their formats are:

POS(str-exp1, str-exp2, int-exp)

and

INSTR(int-exp, str-exp1, str-exp2)

where:

str-exp1 Is the string to be searched.

str-exp2 Is the substring for which you are searching.

int-exp Is the character position where BASIC starts the search.

The position returned by POS and INSTR is relative to the beginning of the string, not the starting position of the search. For example, if you search the string "ABCDE" for the substring "E", it does not matter whether you specify a starting position of one, two, three, four, or five; BASIC still returns the value five as the position where the substring was found.

If the function finds the substring, it returns the position of the substring's first character. Otherwise, it returns zero. For example:

```
10        ALPHA$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
          Z$ = "DEFG"
          X% = POS(ALPHA$,Z$,1%)
          PRINT X%
          Q$ = "TEST"
          Y% = POS(ALPHA$, Q$, 1%)
          PRINT Y%
          END

RUNNH

 4
 0
```

If you specify a starting position other than one, BASIC still returns the position of the substring relative to the beginning of the string. For example:

```
10        ALPHA$ = "ABCDEFGHIJKLMNOP"
          Z$ = "HIJ"
          PRINT POS(ALPHA$, Z$, 7%)
          END

RUNNH

 8
```

If you know that the substring is not near the string's beginning, specifying a starting position greater than one speeds program execution by reducing the number of characters BASIC must search.

You can use the POS function to associate a character string with an integer that you can then use in calculations. This technique is called a table look-up. For example:

```
100 DECLARE STRING CONSTANT TABLE =              &
        "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
    DECLARE STRING MONTH, UPPER_CASE_MONTH, MESSAGE
    DECLARE INTEGER MONTH_LENGTH
    DECLARE REAL MONTH_POS

200 PRINT "Please type the first three letters of a month"
    PRINT "To end the program, type only <RET>";


   Loop_1:

   WHILE 1% = 1%
      INPUT MONTH
      UPPER_CASE_MONTH = EDIT$(MONTH, 32%)
      MONTH_LENGTH = LEN(UPPER_CASE_MONTH)
      EXIT Loop_1 IF MONTH_LENGTH = 0%
      IF MONTH_LENGTH = 3%
        THEN MONTH_POS = (POS(TABLE, UPPER_CASE_MONTH, 1) + 2) / 3
          IF (MONTH_POS = 0%) OR (MONTH_POS <> FIX(MONTH_POS))
               THEN MESSAGE = " Invalid abbreviation, try again"
               ELSE MESSAGE = " is month number" + NUM$(MONTH_POS)
          END IF
        ELSE MESSAGE = " Abbreviation not three characters, try again"
      END IF
```

```
      PRINT MONTH$; MESSAGE

      NEXT

32767 END

RUNNH

Please type the first three letters of a month
To end the program, type only <RET>? Nov
Nov is month number 11
```

This program prompts for a 3–character string, changes the string to uppercase letters and searches the table string to find a match. The WHILE loop executes indefinitely until a carriage return is typed in response to the prompt.

Keep these considerations in mind when you use POS and INSTR:

- If you specify a starting position less than one, POS and INSTR assume a starting position of one.
- If you specify a starting position greater than the searched string's length, POS and INSTR return zero (unless the substring is null).
- When searching for a null string:
  - If you specify a starting position greater than the string's length, POS and INSTR return the string's length plus one.
  - If the string to be searched is also null, POS and INSTR return a value of one.
  - If the specified starting position is less than or equal to one, POS and INSTR return a value of one.
  - If the specified starting position is 1) greater than one and 2) less than or equal to the string's length plus one, POS and INSTR return the specified starting position.

Note that searching for a null string is not the same as searching for the null character. A null string has a length of zero, while the null character has a length of one. The null character is an ASCII character whose value is zero.

4.5.3 SEG$ Function

The SEG$ function extracts a segment (substring) from a string. The original string remains unchanged. Its format is:

SEG$(str-exp, int-exp1, int-exp2)

where:

str-exp Is the input string.

int-exp1 Is the position of the first character extracted.

int-exp2 Is the position of the last character extracted.

SEG$ extracts from the input string the substring that starts at character position int-exp1, up to and including character position int-exp2. It returns the extracted segment. For example:

```
10       PRINT SEG$("ABCDEFG", 3%, 5%)
20       END

RUNNH

CDE
```

If you specify character positions that exist in the string, the length of the returned substring always equals:

(int-exp2 – int-exp1 + 1)

Keep these considerations in mind when you use SEG$:

- If int-exp1 is less than one, BASIC assumes a value of one.
- If int-exp1 is greater than int-exp2 or the length of the string, SEG$ returns a null string.
- If int-exp2 is greater than the length of the string, SEG$ returns all characters from int-exp1 to the end of the string.
- If int-exp1 equals int-exp2, SEG$ returns the character at position int-exp1.

Using the SEG$ function with the string concatenation operator (+), you can replace part of a string. For example:

```
10       A$ = "ABCDEFG"
         C$ = SEG$(A$, 1%, 2%) + "XYZ" + SEG$(A$, 6%, 7%)
         PRINT C$
         PRINT A$
         END

RUNNH

ABXYZFG

ABCDEFG
```

In this example, when BASIC creates C$, it concatenates the first two characters of A$, the 3–letter string XYZ, and the last two characters of A$. The original contents of A$ do not change.

You can use similar string expressions to replace characters in any string. A general formula to replace characters in positions n through m of string A$ with characters in B$ is:

C$ = SEG$(A$,1%,n–1) + B$ + SEG$(A$,m+1,LEN(A$))

For example, to replace the sixth through ninth characters of the string "ABCDEFGHIJK" with "123456", enter:

```
10       A$ = "ABCDEFGHIJK"
         B$ = "123456"
         C$ = SEG$(A$,1%,5%) + B$ + SEG$(A$,10%,LEN(A$))
         PRINT C$
         PRINT A$
         PRINT B$
         END
```

```
RUNNH

ABCDE123456JK
ABCDEFGHIJK
123456
```

The following formulas are more specific applications of the general formula.

- To replace the first n characters of A$ with B$:

  C$ = B$ + SEG$(A$,n+1,LEN(A$))

- To replace all but the first n characters of A$ with B$:

  C$ = SEG$(A$,1,n) + B$

- To replace all but the last n characters of A$ with B$:

  C$ = B$ + SEG$(A$,(LEN(A$)–n) + 1, LEN(A$))

- To replace the last n characters of A$ with B$:

  C$ = SEG$(A$,1,LEN(A$)–n) + B$

- To insert B$ in A$ after the nth character in A$:

  C$ = SEG$(A$,1,n) + B$ + SEG$(A$,n+1,LEN(A$))

4.5.4 MID$ Function

The MID$ function extracts a substring from a string by specifying the substring's starting position and length. The original string remains unchanged. The format for MID$ is:

MID$(str-exp, int-exp1, int-exp2)

where:

str-exp Is the input string.

int-exp1 Is the first character position of the substring.

int-exp2 Is the number of characters in the substring.

MID$ extracts from the input string the substring that starts at character position int-exp1, and includes a total of int-exp2 characters. It returns the extracted segment. For example:

```
10      ALPHA$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20      PRINT MID$(ALPHA$, 15%, 5%)
30      PRINT MID$("ENCYCLOPEDIA", 3%, 6%)
40      END

RUNNH

OPQRS
CYCLOP
```

Keep these considerations in mind when you use MID$:

- If int-exp1 (the substring's first character position) is greater than the string expression's length, MID$ returns a null string.

- If int-exp1 is less than one, BASIC assumes a starting position of one.
- If int-exp2 (the length of the substring) is less than or equal to zero, MID$ returns a null string.
- If int-exp2 is greater than the number of characters remaining in the string, MID$ returns all the characters remaining in the string.

Instead of using MID$, you can perform the same operation with SEG$:

```
MID$(A$, B%, C%)
```

This is equivalent to:

```
SEG$(A$, B%, B% + C% – 1%)
```

4.5.5 LEFT$ Function

The LEFT$ function lets you extract substrings from the left side of a character string. Its format is:

```
LEFT$(str-exp, int-exp)
```

where:

str-exp Is the input string.

int-exp Is the position of last character extracted, counting from the left side of the input string.

For example:

```
100     LINPUT "File specification? (Do not type a network node)"; FILESPEC$
        COLONPOS = POS(FILESPEC$, ':', 1)
        IF COLONPOS = 0
        THEN
            PRINT "No disk in that file spec."
        ELSE
            PRINT "That file is on ";LEFT$(FILESPEC$, COLONPOS)
        END IF
```

This program accepts a file specification as an input string and uses the POS and LEFT$ functions to extract the disk device.

These restrictions apply to the LEFT$ function:

- If the integer expression is greater than the string's length, LEFT$ returns the entire string.
- If the integer expression is less than or equal to zero, LEFT$ returns a null string.

Instead of using LEFT$, you can use SEG$ to perform the same operation:

```
LEFT$(A$, B%)
```

This is equivalent to:

```
SEG$(A$, 1%, B%)
```

4.5.6 RIGHT$ Function

The RIGHT$ function extracts a substring from the right side of a string. Its format is:

RIGHT$(str-exp, int-exp)

where:

str-exp Is the input string.

int-exp Is the position of the first character extracted, counting from the left side of the input string.

BASIC extracts from str-exp the character at the specified position and all characters to the right. For example:

```
10      PRINT RIGHT$("ABCDEFG",3%)
20      END

RUNNH

CDEFG
```

RIGHT$ starts at the third character position of "ABCDEFG" and extracts this character and all others to the end of the string.

These restrictions apply to the RIGHT$ function:

- If int-exp (the substring's first character position) is greater than the string's length, RIGHT$ returns a null string.
- If int-exp is less than or equal to zero, RIGHT$ returns the entire string.

Instead of using RIGHT$, you can use SEG$ to perform the same operation:

RIGHT$(A$, B%)

This is equivalent to:

SEG$(A$, B%, LEN(A$))

4.5.7 STRING$ Function

The STRING$ function creates a character string containing multiple occurrences of a single character. Its format is:

STRING$(int-exp1, int-exp2)

where:

int-exp1 Is the length of the returned string.

int-exp2 Is the ASCII value of the character that makes up the string. This value is treated modulo 256.

For example:

```
10       OUT$ = STRING$(10%, 65%)
20       PRINT OUT$
30       END

RUNNH

AAAAAAAAAA
```

This program creates a 10–character string containing uppercase As, which have ASCII value 65.

Keep these considerations in mind when you use the STRING$ function:

- If int-exp1 (the length of the returned string) is less than or equal to zero, STRING$ returns a null string.
- If int-exp1 is greater than 65535 (in VAX–11 BASIC) or 32767 (in PDP–11 BASIC–PLUS–2), BASIC signals an error.

4.5.8 SPACE$ Function

The SPACE$ function creates a character string containing spaces. Its format is:

SPACE$(int-exp)

where:

int-exp Is the number of spaces in the string.

For example:

```
10       A$ = "ABC"
         B$ = "XYZ"
         PRINT A$ + SPACE$(3%) + B$
         END

RUNNH

ABC    XYZ
```

4.5.9 TRM$

The TRM$ function removes trailing blanks and tabs from a string. The input string remains unchanged. The format for TRM$ is:

TRM$(str-exp)

where:

str-exp Is any string expression.

For example:

```
10       A$ = "ABCDE    "
20       B$ = "XYZ"
30       FIRST$ = A$ + B$
40       SECOND$ = TRM$(A$) + B$
50       PRINT FIRST$
60       PRINT SECOND$
70       END
```

```
RUNNH

ABCDE    XYZ
ABCDEXYZ
```

The TRM$ function is especially useful for extracting the nonblank characters from a fixed-length string (for example, a COMMON or MAP, or a parameter passed from a program written in another language).

4.5.10 EDIT$

The EDIT$ function performs one or more string editing functions, depending on the value of an argument you supply. The input string remains unchanged. The format for EDIT$ is:

```
EDIT$(str-exp, int-exp)
```

where:

str-exp Is any string expression.

int-exp Determines the editing function performed.

Table 4–2 shows the action BASIC takes for a given value of int-exp.

Table 4–2: EDIT$ Options

Value of int-exp	Effect
1	Discards each character's parity bit (bit 7). Note that you should not use this value for characters in the DEC Multinational Character Set.
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.
64	Converts left brackets ([) to left parentheses [(], and right brackets (]) to right parentheses [)].
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.

All values are additive; for example, by specifying 168, you can:

- Discard leading spaces and tabs (value 8)
- Convert lowercase letters to uppercase (value 32)
- Discard trailing spaces and tabs (value 128)

The following program requests an input string, discards all spaces and tabs, converts lowercase letters to uppercase, and converts brackets to parentheses:

```
10      LINPUT "PLEASE TYPE A STRING";INPUT_STRING$
        NEW_STRING$ = EDIT$(INPUT_STRING$, 2% + 32% + 64%)
        PRINT NEW_STRING$
        END

RUNNH

PLEASE TYPE A STRING? 88  abc<TAB>[5,5]
88ABC(5,5)
```

4.6 Manipulating String Data with Multiple MAPs

Mapping a string storage area in more than one way lets you: 1) extract a substring from a string or 2) concatenate strings. For example:

```
10 MAP (EMPREC) FIRST_NAME$ = 10,                    &
                LAST_NAME$ = 20,                     &
                STREET_NUMBER$ = 6,                  &
                STREET$ = 15,                        &
                CITY$ = 20,                          &
                STATE$ = 2,                          &
                ZIP$ = 5,                            &
                WAGE_CLASS$ = 2,                     &
                DATE_OF_REVIEW$ = 8,                 &
                SALARY_YTD$ = 10,                    &
                TAX_YTD$ = 10
   MAP (EMPREC) FULL_NAME$ = 30,                     &
                ADDRESS$ = 48,                       &
                SALARY_INFO$ = 30
   MAP (EMPREC) EMPLOYEE_RECORD$ = 108
```

These three MAP statements reference the same 108 bytes of data. You can move data into this MAP in different ways. For example, you can use READ and DATA statements:

```
100 READ EMPLOYEE_RECORD$
110 DATA "WILLIAM   DAVIDSON                2241  MADISON BLVD    " &
"SCRANTON            PA14225A912/10/78$13,325.77$925.31"
```

Because all the MAP statements in this example reference the same storage area (EMPREC), you can access parts of this area through the mapped variables. Consider these examples:

```
200 PRINT FULL_NAME$
210 PRINT WAGE_CLASS$
220 PRINT SALARY_YTD$
```

This produces:

```
WILLIAM   DAVIDSON
A9
$13,325.77
```

```
300 PRINT LAST_NAME$
310 PRINT TAX_YTD$
```

This produces:

```
DAVIDSON
$925.31
```

You can assign a new value to any of the mapped variables. For example:

```
500           Loop_1:
              WHILE 1% = 1%
                   INPUT "Changes? (please type YES or NO)"; CH$
                   EXIT Loop_1 IF CH$ = "NO"
                   PRINT "1.  FIRST NAME"
                   PRINT "2.  LAST NAME"
                   PRINT "3.  STREET NUMBER"
                   PRINT "4.  STREET"
                   PRINT "5.  CITY"
                   PRINT "6.  STATE"
                   PRINT "7.  ZIP"
                   PRINT "8.  WAGE CLASS"
                   PRINT "9.  DATE OF REVIEW"
                   PRINT "10. SALARY YTD"
                   PRINT "11. TAX YTD"
                   INPUT "CHANGE NUMBER"; NUMBER%

                   SELECT NUMBER%
                   CASE 1%
                        INPUT "FIRST NAME"; FIRST_NAME$
                   CASE 2%
                        INPUT "LAST NAME"; LAST_NAME$
                   CASE 3%
                        INPUT "STREET NUMBER"; STREET_NUMBER$
                   CASE 4%
                        INPUT "STREET"; STREET$
                   CASE 5%
                        INPUT "CITY"; CITY$
                   CASE 6%
                        INPUT "STATE"; STATE$
                   CASE 7%
                        INPUT "ZIP CODE"; ZIP$
                   CASE 8%
                        INPUT "WAGE CLASS"; WAGE_CLASS$
                   CASE 9%
                        INPUT "DATE OF REVIEW"; DATE_OF_REVIEW$
                   CASE 10%
                        INPUT "SALARY YTD"; SALARY_YTD$
                   CASE 11%
                        INPUT "TAX YTD"; TAX_YTD$
                   CASE ELSE
                        PRINT "Invalid choice"
                   END SELECT
              NEXT
32767         END
```

This produces:

```
Changes? (please type YES or NO)? YES
1.  FIRST NAME
2.  LAST NAME
3.  STREET NUMBER
4.  STREET
5.  CITY
6.  STATE
7.  ZIP CODE
8.  WAGE CLASS
```

(continued on next page)

```
9.  DATE OF REVIEW
10. SALARY YTD
11. TAX YTD

CHANGE NUMBER? 10
SALARY YTD? 14,277.08
Changes? (Please type YES or NO)? YES
CHANGE NUMBER? 11
TAX YTD? 998.32
Changes? (Please type YES or NO)? NO
```

When you modify a mapped variable, you modify all other variables that overlay the same storage. For example, after changing the variables SALARY_YTD$ and TAX_YTD$, the statement:

```
10000 PRINT EMPLOYEE_RECORD$
```

produces:

```
WILLIAM   DAVIDSON            2241  MADISON BLVD   SCRANTON
PA14225A912/10/78$14,277.08$998.32
```

You can also move data into a MAP using terminal input, arrays, and files. See Chapter 5 for more information on the MAP statement.

4.7 Manipulating Strings with the CHANGE Statement

The CHANGE statement: 1) converts a string of characters to an array containing their numeric ASCII values or 2) converts a list of numbers to a string of ASCII characters. Its format is:

CHANGE str-exp TO num-array

CHANGE num-array TO str-vbl

where:

str-exp Is any string expression.

num-array Is a one- or two-dimensional numeric array.

In the first case, when the CHANGE statement is executed, BASIC stores the length of the string expression in element zero of the array. In the second case, element zero of the array must contain the length of the string. If the value in element zero is zero, BASIC creates a zero-length, or null, string.

You should use one-dimensional integer arrays when using the CHANGE statement. If you use arrays of more than one dimension, BASIC accesses only the array's first row.

The CHANGE statement lets you associate a character position with an array subscript. This is useful in searching a string for a particular character. For example:

```
100     DIM INTEGER SEARCH(80%)
200     PRINT "This program accepts a string of"
        PRINT "up to 80 characters and returns"
        PRINT "the character position of the first"
        PRINT "occurrence of the character 'f'."
300     INPUT "String"; A$
400     CHANGE A$ TO SEARCH
```

```
Loop_1:

        FOR I% = 1% TO SEARCH(0%)    !SEARCH(0) contains length of string
            IF SEARCH(I%) = 102%    !102 is the ASCII value of "f"
                THEN PRINT "'f' was found at position";I%
                     EXIT Loop_1
            END IF
600     NEXT I%
700     IF (I% = SEARCH(0%)) AND (SEARCH(I%) <> 102%)
            THEN PRINT "'f' was not found in input string"
1000    END
```

The CHANGE statement also makes it easy to change characters within a string. For example:

```
100     INPUT "String"; A$
200     DIM INTEGER ARRAY(80%)
300     CHANGE A$ TO ARRAY
400     FOR I% = 1% TO ARRAY(0%)
            IF (97% <= ARRAY(I%)) AND (ARRAY(I%) <= 122%)
                THEN ARRAY(I%) = ARRAY(I%) - 32%
500     NEXT I%
600     CHANGE ARRAY TO A$
700     PRINT A$
```

This program changes lowercase letters to uppercase. It first accepts an input string and changes it to an integer array. The FOR statement specifies a starting value of 1 (the first array element) and an ending value of ARRAY(0) (because this element contains the length of the string).

In the FOR/NEXT loop, the program tests each array element for values between 97 and 122, inclusive (these are the ASCII values for lowercase letters). When an array element with one of these values is found, 32 is subtracted from it. Thus, the value 97 (lowercase "a") is changed to 65 (uppercase "A").

When the entire string has been processed, the CHANGE statement in line 600 converts the array back to a string. Note that ARRAY(0) still contains the string length assigned by the CHANGE statement in line 300; therefore, the string length does not change.

4.8 String Virtual Arrays

Virtual arrays are stored on disk. You create a virtual array by opening a disk file and then using the DIM # statement to dimension the array on the open channel. This section describes only string virtual arrays. See Chapter 9 for more information on virtual arrays.

Elements of string virtual arrays behave much like dynamic strings, except that:

- When you create the virtual string array, you specify a maximum length for the array's elements. The length of an array element can never exceed this maximum. If you do not supply a length, the default is 16 characters.

- A string virtual array element cannot contain trailing nulls.

When you assign a value to a string virtual array element, BASIC pads the value with nulls, if necessary, to fit the length of the virtual array element. However, when you retrieve the virtual array

element, BASIC strips all trailing nulls from the string. Therefore, when you access an element in a virtual string array, the string never has trailing nulls. For example:

```
10      DIM #1%, STRING TEST(5)
20      OPEN "TEST" AS FILE #1%, ORGANIZATION VIRTUAL
30      A$, TEST(1%) = "ABCDE" + STRING$(5%, 0%)
40      PRINT "LENGTH OF A$ IS: "; LEN(A$)
50      PRINT "LENGTH OF TEST(1) IS: "; LEN(TEST(1%))
60      END

RUNNH

LENGTH OF A$ IS: 10
LENGTH OF TEST(1) IS: 5
```

Lines 10 and 20 dimension a virtual string array and open a file on channel 1. Line 30 assigns a 10–character string to the first element of this string array, and to the variable A$. This 10–character string consists of "ABCDE" plus five null characters. The PRINT statements in lines 40 and 50 show that the length of A$ is 10, while the length of TEST(1) is only five because BASIC strips trailing nulls from string array elements.

Although the storage for virtual string array elements is fixed, the length of a string array element can change because BASIC strips the trailing nulls whenever it retrieves a value from the array.

Chapter 5
Data Definition

This chapter describes how you explicitly assign data types to program variables and how to allocate and use data storage.

5.1 Declarative Statements

Declarative statements are those that define objects in a BASIC program. Objects can be variables, arrays, constants, and user-defined functions within a program module. They can also be routines, variables, and constants external to the program module. Declarative statements always assign names to the objects declared and usually assign other attributes such as a data type to them.

On VAX–11 systems, declarative statements can also be used with user-defined data types (RECORD statements). See *BASIC on VAX/VMS Systems* for more information on the RECORD statement.

You use declarative statements to assign data types to: 1) variables, arrays, and named constants and 2) the values returned by functions. By declaring the objects used in your program, you make the program much easier to understand, modify, and debug.

5.2 Data Types

At its most fundamental level, a data type is a format for information storage. All information is stored in the computer as bit patterns (groups of ones and zeros). Data types specify that the computer should interpret these patterns in a certain way.

BASIC programs allow four general data types: integer, floating-point, string, and packed decimal (VAX–11 BASIC only). Each of these general data types has unique characteristics that determine the way you use it. For example, integers are useful for numeric computations involving whole numbers, and strings provide a way to manipulate alphanumeric characters.

Within integer and floating-point data types there are further subdivisions. For example, integers can be classed as BYTE, WORD, and LONG. Choosing one of these integer subdivisions lets you control two things: 1) the amount of storage required for the integer and 2) the range of values that the integer can accept. See Table 5–1 for more information on the range and storage requirements of these integer subtypes.

Similarly, floating-point data can be classed as SINGLE, DOUBLE, GFLOAT (VAX–11 BASIC only), and HFLOAT (VAX–11 BASIC only). See Table 5–1 for more information on the range and storage requirements of these floating-point subtypes. The choice you make when assigning numeric data types always involves a tradeoff between storage requirements and precision or range.

In addition to numeric and string data types, BASIC also provides a unique data type called RFA. Variables of the RFA data type require six bytes of storage and can contain only a Record File Address. RFA variables are used with RMS file I/O and the operations that can be performed on them are strictly limited. See Chapter 9 for more information on the RFA data type.

Traditionally, BASIC programs have had just three data types: integer, string, and floating-point. You assigned a data type to a variable by adding a suffix to the variable names; a dollar sign ($) denoted a string variable, a percent sign (%) denoted an integer variable, and variable names without suffixes denoted floating-point variables.

BASIC now lets you explicitly assign data types to variables, parameters, and functions. This feature gives you more control over the storage and precision used by your program. You can still use implicit data typing in your programs; however, you should not mix explicit and implicit data typing in the same statement. You can ensure that all program variables are explicitly declared by using the /TYPE=EXPLICIT qualifier when you compile your programs, or by specifying explicit data typing with the OPTION statement. See Section 5.3 for more information on the OPTION statement.

Table 5–1 lists the keywords you use to assign data types and their meanings.

Table 5–1: 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.

This chapter describes the BASIC statements that explicitly type data (DECLARE, EXTERNAL, FUNCTION, COMMON, DEF, DIMENSION, MAP, SUB, and RECORD (in VAX–11 BASIC)) and also describes the syntax for explicitly typed constants.

As shown in the table, there are four data-type keywords that specify integer data. Data-type INTEGER is a general data-type because it specifies only that a variable contains integer data. Data-types BYTE, WORD, and LONG specify exactly how much storage is allocated to an integer variable. If you specify data-type INTEGER, the subtype of integer variables depends on the default integer data-type in effect when the program is compiled. This default is determined by:

- The qualifier you use to compile the program.
- The program's OPTION statement, if present. See Section 5.3 for more information on the OPTION statement.

Similarly, there are five data-type keywords that specify floating-point data. Data-type REAL is a general data type because it specifies only that a variable contains floating-point data. Data-types SINGLE, DOUBLE, GFLOAT, and HFLOAT specify exactly how much storage is allocated to a floating-point variable. If you specify data-type REAL, the subtype of floating-point variables depends on the default floating-point subtype in effect when the program is compiled. This default is determined by:

- The qualifier you use to compile the program.
- The OPTION statement, if present. See Section 5.3 for more information on the OPTION statement.

In this manual, BYTE, WORD, and LONG are referred to as subtypes of the INTEGER data type. SINGLE, DOUBLE, GFLOAT, and HFLOAT are referred to as subtypes of the REAL data type.

It is important to note that choosing a numeric subtype always involves a tradeoff between storage requirements and range or precision. You can reduce the size of an executable image by choosing the smallest numeric subtype that is large enough to meet your needs.

5.3 Setting the Default Data Type and Size

There are two ways to set the default data type and size for your program. These are:

- With qualifiers:
  - /TYPE_DEFAULT
  - /BYTE, /WORD, or /LONG
  - /SINGLE, /DOUBLE, /GFLOAT (VAX–11 BASIC only), or /HFLOAT (VAX–11 BASIC only)
  - /DECIMAL_SIZE (VAX–11 BASIC only)
- With the OPTION statement

See *BASIC on VAX/VMS Systems*, *BASIC on RSX–11M/M–PLUS Systems*, or *BASIC on RSTS/E Systems* for more information on qualifiers.

The OPTION statement also sets the default data type and size and can override the defaults set with qualifiers. When using OPTION to set the default type and size, its format is:

```
OPTION option-clause,...

option-clause:  { TYPE = type-clause                                  }
                { SIZE = size-clause                                  }
                { SCALE = int-const                                   }
                { { ACTIVE   } = active-clause   (VAX-11 BASIC only)  }
                { { INACTIVE } = active-clause   (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             }  (all VAX-11 BASIC only)
                { DECIMAL ROUNDING  }
                { SUBSCRIPT CHECKING }
```

You can have more than one OPTION statement in a program module; however, OPTION statements can be preceded only by a SUB, FUNCTION, REM, or another OPTION statement.

Note that VAX–11 BASIC, the OPTION statement can also specify:

- Integer and packed decimal overflow checking
- Program optimization
- Rounding or truncation of packed decimal numbers
- Subscript checking

See the *BASIC Reference Manual* for more information about OPTION.

This is an example of the OPTION statement:

```
1 OPTION TYPE = EXPLICIT,        ! Variables must be declared    &
         SIZE = INTEGER WORD,    ! 16-bit integers               &
         SIZE = REAL DOUBLE      ! 64-bit floating-point numbers &
                                 ! by default
```

This statement specifies that:

- All program variables must be explicitly typed
- Any variable typed as INTEGER is a WORD integer
- Any variable typed as REAL is a DOUBLE floating-point number

You can create variables of other data types by explicitly declaring them with the DECLARE statement.

5.4 Explicitly Declaring Variables (DECLARE Statement)

The DECLARE statement explicitly assigns a data type or subtype to a variable, function, or constant. The format for declaring variables is:

```
DECLARE data-type vbl [, [data-type] vbl]. . .
```

where:

data-type Is a subtype optionally preceded by a general data type or a general data type.

vbl Is a variable or array name that does not end in a percent or dollar sign.

See Table 5–1 for data-type keywords.

The subtype you specify overrides any defaults specified in the BASIC environment or with the /INTEGER_SIZE, /REAL_SIZE, and /DECIMAL_SIZE qualifiers. For example, if you compile your program with the /INTEGER_SIZE=WORD qualifier and DECLARE an integer variable to be LONG, the variable is LONG rather than WORD. In this format of the DECLARE statement, data-type STRING specifies a dynamic string variable.

Note

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 statement that explicitly names a data type.

You can define a variable only once in a program. For example, if a variable name appears in a DECLARE statement, it cannot also appear in a COMMON or MAP statement.

For example:

```
200             DECLARE INTEGER         &
                                        ALPHA, &
                                        BETA, &
                                        GAMMA
300             DECLARE LONG            &
                                        LAMBDA, &
                                        EPSILON, &
                                        CHI
400             DECLARE REAL            &
                                        DELTA, &
                                        IOTA, &
                                        RHO
500             DECLARE GFLOAT          &
                                        OMICRON, &
                                        XI, &
                                        TAU
```

The DECLARE statement in line 200 specifies only that variables ALPHA, BETA, and GAMMA are integers of the default INTEGER subtype. The integer subtype can be specified with the /INTEGER_SIZE qualifier or with the OPTION statement.

The DECLARE statement in line 300 specifies that variables LAMBDA, EPSILON, and CHI are LONG integers, overriding any default integer subtype.

Similarly, the DECLARE statement in line 400 specifies only that variables DELTA, IOTA, and RHO are floating-point numbers of the default REAL subtype. The REAL subtype can be specified with the /REAL_SIZE qualifier or with the OPTION statement.

The DECLARE statement in line 500 specifies that variables OMICRON, XI, and TAU are GFLOAT floating-point variables, overriding any default REAL subtype.

You should use unique variable names to avoid confusion and make program documentation easier. For example, if you DECLARE variable B to be LONG, there cannot also be a floating-point variable B in your program. It is possible to have both an implicit integer variable B% and an explicit INTEGER variable B in the same program; however, this is poor programming practice.

You can also use the DECLARE statement to assign a data type and value to DEF functions and constants. See Section 5.5 for an explanation of declaring named constants. The format for declaring DEF functions is:

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

where:

def-nam Is the name of the DEF function. Function names conform to the same rules as variable names; however, the DECLARE statement does not allow names ending with a percent or dollar sign.

def-parm Is a BASIC data-type keyword. When you actually invoke the DEF function, BASIC changes the actual arguments to match the specified data type, if necessary.

DECLARE FUNCTION lets you: 1) assign a data type to parameters and to the value a function returns and 2) name the function without using the usual convention (beginning the function name with FN and ending the function name with a percent or dollar sign suffix). For example:

```
100 DECLARE STRING FUNCTION CONCAT (STRING, STRING) !Declare the function
.
.
.

1000 DEF CONCAT (STRING Y, STRING Z)        !Define the function
1100 CONCAT = Y + Z
1200 END DEF
.
.
.
20000 NEW_STRING$ = CONCAT(A$, B$)          !Invoke the function
.
.
.

32767 END
```

This format allows only one data type in a single statement. Declaring more than one type of function requires more than one DECLARE statement.

These new data typing features give you more control over storage allocation. Compiling a program with /TYPE_DEFAULT=EXPLICIT is particularly useful because it causes BASIC to signal an error when an implicit variable is encountered. This prevents a typing mistake from being interpreted as a new variable. DIGITAL supports implicit variables for compatibility with other BASICs and also because they are useful for beginning programmers. However, DIGITAL recommends that you use explicit declarations for new program development.

5.5 Explicitly Named Constants

Constants are values that do not change during program execution. You can declare named constants within a program unit with the DECLARE statement. You can also refer to constants outside the program unit with the EXTERNAL statement. In addition, VAX–11 BASIC provides notation for binary, octal, decimal, and hexadecimal constants.

Named constants are useful because:

1. If a commonly-used constant must be changed, you can make the change in a single place.

2. They make the program easier to understand.

5.5.1 Declaring Constants Within a Program Unit

The format for declaring constants within a program unit is:

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

where:

data-type Is any valid BASIC data type.

const-nam Is the name of the constant.

value Is a literal, a previously declared constant, a predefined constant, or an expression containing literals and previously defined constants.

In PDP–11 BASIC–PLUS–2, you can specify only a single value for floating-point named constants. That is, the value assigned to named floating-point constants cannot be an expression.

Note that the value assigned to the named constant need not be in the allowable range of the default data type; however, it must be in the valid range of the data type being declared. For example:

```
10      DECLARE LONG CONSTANT XYZ = 1000%
```

Line 10 declares a LONG constant named XYZ and assigns it a value of 1000. In DECLARE CONSTANT statements, BASIC signals an overflow error only if the value is outside the range of the data type being declared.

This example declares a double-precision constant:

```
100 DECLARE DOUBLE CONSTANT PLANCKS = 6.6237E-27
200 INPUT "FREQUENCY"; FREQ
300 PRINT "ENERGY EQUALS"; PLANCKS / FREQ
400 END
```

A DECLARE CONSTANT statement allows only one data type. To declare a constant of a different data type, use a second DECLARE CONSTANT statement.

5.5.2 Declaring Constants External to the Program Unit

To declare constants external to the program unit, use the EXTERNAL statement. Its format is:

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

where:

data-type Is a data-type keyword. VAX–11 BASIC allows external constants of data-type BYTE, WORD, LONG, INTEGER, SINGLE, or REAL (if the default size is SINGLE). BASIC–PLUS–2 allows external constants of data-type WORD or INTEGER (if the default size is WORD).

const-nam Is the name of the constant. On VAX/VMS systems, this might be a system status code, or a global constant declared in a VAX–11 MACRO or VAX–11 BLISS program. On PDP–11 systems, this can be an RMS symbol or a global constant declared in a MACRO–11 or FORTRAN program.

The linker or task builder automatically supplies the values for constants specified in EXTERNAL statements.

5.5.3 Explicitly Formed Literals

By using a special notation, BASIC lets you specify the value and data type of numeric and string literals. These explicitly formed literals are useful when:

- Passing numeric literals as parameters
- Specifying a numeric literal whose value is outside the allowable range of the default data type

The format of this notation is:

<quote>numeric-string<quote>subtype-abbr

where:

quote Is either a single or double quotation mark.

numeric-string Is a string composed of digits, and an optional decimal point. Numeric-string can also be a number in E format. However, a leading minus sign cannot appear inside the quotes.

subtype-abbr Is an abbreviation for a data-type keyword. You cannot use general numeric data types INTEGER and REAL; you must use a specific subtype abbreviation.

Subtype abbreviations are:

B	BYTE
W	WORD
L	LONGWORD
F	SINGLE
D	DOUBLE
G	GFLOAT (VAX–11 BASIC only)
H	HFLOAT (VAX–11 BASIC only)
P	DECIMAL (VAX–11 BASIC only)

These are examples or numeric literals:

"40000"L	Specifies a LONG constant with a value of 40000.
"32767"W	Specifies a WORD constant with a value of 32767.
".000000000001"D	Specifies a DOUBLE constant with a value of 10^{-12}.
"10E5"F	Specifies a SINGLE constant with a value of 100000.

The value of a numeric literal must be within the allowable range of the default data type, unless you use this notation. For example:

```
100     DECLARE WORD A, B, C
200     A = 500
```

If this program is compiled with /INTEGER_SIZE=BYTE, VAX–11 BASIC reports "Integer error or overflow" at line 200 because the value 500 is outside the range of the default container size. To obtain the desired effect, you should use explicit numeric literal notation:

```
100     DECLARE WORD A, B, C
200     A = "500"W
```

Note that this constant notation lets you specify values outside the range of the default data type.

In VAX–11 BASIC, you can also specify a radix for integer literals. The radix can be base 10 (decimal), base 2 (binary), base 8 (octal), and base 16 (hexadecimal). This feature is useful in forming conditional expressions and in using logical operations, because it provides a means to set or clear individual bits in the representation of an integer.

The format for specifying these radices is:

radix"numeric-string"subtype-abbr

where:

radix	Is: D (decimal), B (binary), O (octal), or X (hexadecimal).
numeric-string	Is a string composed of digits. For binary numbers, numeric-string can contain the digits 0 and 1. For octal numbers, numeric-string can contain the digits 0 through 7. For hexadecimal numbers, numeric-string can contain the digits 0 through 9 and letters A through F.
subtype-abbr	Is an abbreviation for an integer subtype: B (BYTE), W (WORD), or L (LONG).

Note

In VAX–11 BASIC, a "D" can appear in both the radix position and the subtype position. The "D" that precedes the numeric string specifies decimal radix, not packed decimal data type. You specify a double-precision floating-point constant with a D in the subtype position. You specify a packed decimal constant with a P in the subtype position.

It is important to understand that, when you specify a radix other than decimal, BASIC treats the numeric string as an unsigned integer. However, if this value is assigned to a variable or used in an expression, BASIC treats the variable as a signed integer. For example:

```
100       DECLARE WORD A
          A = B'1111111111111111'W
          PRINT A
RUNNH

 -1
```

In this example, BASIC assigns a value to the 16–bit storage location of variable A by setting all the bits in that location. While the value of 1111111111111111 in base 2 is actually 65535, this number is not representable in a WORD integer because integers are represented in *two's complement* notation. Thus, when BASIC executes the PRINT statement, variable A is displayed as a signed integer. Because the two's complement representation of –1 is 1111111111111111, BASIC prints a –1.

BASIC also lets you represent a single-character string in terms of its 8–bit ASCII value. The format is:

radix"numeric-string"C

where:

radix	Is: D (decimal), B (binary), O (octal), or X (hexadecimal).
numeric-string	Is a string composed of digits. For binary numbers, numeric-string can contain the digits 0 and 1. For octal numbers, numeric-string can contain the digits 0 through 7. For hexadecimal numbers, numeric-string can contain the digits 0 through 9 and letters A through F. Note that the value of the numeric string must be between 0 and 255, inclusive.
C	Is a subtype abbreviation meaning CHARACTER.

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

```
100 DECLARE STRING CONSTANT CTRL_S = X'13'C, CTRL_Q = X'11'C
```

This program declares two string constants named CTRL_S (ASCII hexadecimal value 13) and CTRL_Q (ASCII hexadecimal value 11).

5.6 Operations Using Multiple Data Types

When an expression contains operands of different data types, it is called a *mixed-mode* expression. Before a mixed-mode expression can be evaluated, the operands must be converted, or *promoted*, to a common data type. The result of the evaluation may also be converted depending on the data type of the variable to which it is assigned.

When evaluating mixed-mode expressions, BASIC performs these promotions such that no operand loses any range or precision. When assigning values to variables, BASIC converts the result of the expression to the data type of the variable. If the value of the expression is outside the allowable range of the variable's data type, BASIC signals "Integer error or overflow", "Floating-point error or overflow", or "DECIMAL error or overflow".

In general, BASIC promotes operands with different data-types to the lowest data type that can hold the largest and most precise possible value of either operand's data type. BASIC then performs the operation in that data-type, and yields a result of that data type. If the result of the expression is assigned to a variable, BASIC converts the result to the data-type of the variable. Table 5–2 lists the resulting data type for all combinations except those involving DECIMAL (VAX–11 BASIC only) data types.

Table 5–2: Result Data Types in BASIC Expressions

Operand 1	Operand 2					**VAX–11 BASIC**	
	BYTE	WORD	LONG	SINGLE	DOUBLE	GFLOAT	HFLOAT
BYTE	BYTE	WORD	LONG	SINGLE	DOUBLE	GFLOAT	HFLOAT
WORD	WORD	WORD	LONG	SINGLE	DOUBLE	GFLOAT	HFLOAT
LONG	LONG	LONG	LONG	SINGLE	DOUBLE	GFLOAT	HFLOAT
SINGLE	SINGLE	SINGLE	SINGLE	SINGLE	DOUBLE	GFLOAT	HFLOAT
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	HFLOAT	HFLOAT
GFLOAT	GFLOAT	GFLOAT	GFLOAT	GFLOAT	HFLOAT	GFLOAT	HFLOAT
HFLOAT	HFLOAT	HFLOAT	HFLOAT	HFLOAT	HFLOAT	HFLOAT	HFLOAT

Note

GFLOAT and HFLOAT are VAX–11 BASIC data types only.

As Table 5–2 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 hold the largest and most precise possible SINGLE value. If BASIC did not promote the SINGLE value and the operation yielded a more precise result than was representable in SINGLE, loss of precision would occur.

With one exception, the resulting data type is the same as that of the operand with the higher data type. The exception is when the operands are DOUBLE and GFLOAT. When an expression contains a DOUBLE and a GFLOAT operand, BASIC promotes both values to HFLOAT, and returns an HFLOAT value. This 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.

If all operands in an expression are BYTE, WORD, or SINGLE, all promotions remain within that set of data types. If all operands in an expression are BYTE, WORD, LONG, or DOUBLE, all promotions remain within that set of data types.

VAX–11 BASIC also allows the DECIMAL(d,s) data type. DECIMAL values are converted to REAL before exponentiation. For all other operations involving a DECIMAL value, the number of digits (d) and the scale or position of the decimal point (s) in the result depend 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.

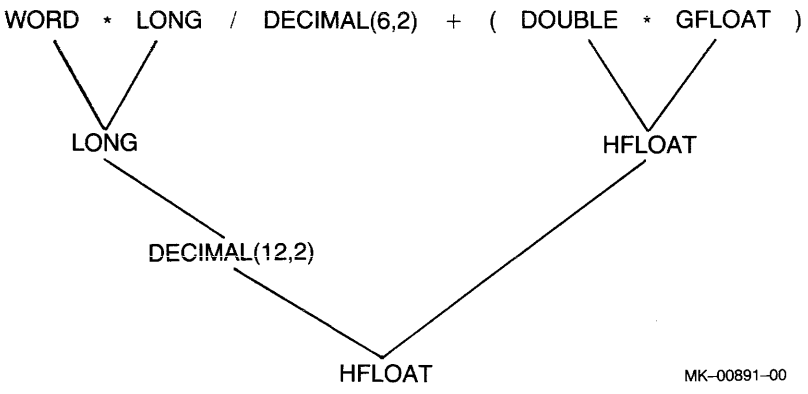
Note that only INTEGER data types are converted to the DECIMAL data type. If one operand is DECIMAL and one is floating-point, the DECIMAL value is converted to a floating-point value. The total number of digits (d) in the DECIMAL value determines its new data type:

Range of d	Converted to:
<=1 through <= 6	SINGLE
<= 7 through <= 15	DOUBLE GFLOAT HFLOAT
= 16	DOUBLE
< 17 through <=31	HFLOAT

If the value of d is between 7 and 15, the operand is converted to DOUBLE if the floating-point operand is DOUBLE, to GFLOAT if the floating-point operand is GFLOAT, and to 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.

Figure 5–1 shows a mixed-mode expression, and the data types of the intermediate and final results.

Figure 5–1: Mixed-mode Expression Results



Note that the LONG integer is first converted to DECIMAL(10,0). When BASIC performs the division, both operands are converted to DECIMAL(12,2).

You can convert any numeric variable or expression to a specified data type with the REAL, INTEGER, and DECIMAL functions. See the *BASIC Reference Manual* for more information.

5.7 Allocating Static Storage

BASIC programs allocate both static and dynamic storage. The size of static storage does not change during program execution. Variables and arrays appearing in MAP or COMMON statements use static storage. Because this storage is static, all string variables appearing in MAPs or COMMONs are fixed-length strings.

Dynamic storage is allocated when the program executes. Variables and arrays declared in the following statements use dynamic storage:

- DECLARE statements
- DIMENSION statements
- Implicitly declared variables

Normally, string variables and arrays declared in this way are dynamic strings, and their length can change during program execution. However, if you declare or dimension an array of a user-defined data type (a RECORD name), then all string variables and arrays are fixed-length strings. See *BASIC on VAX/VMS Systems* for more information about the RECORD statement.

MAP and COMMON statements create a named storage area called a program section (PSECT). MAP statements require a map name, but in COMMON statements the name is optional. The PSECT name is the same as the MAP or COMMON name. If you do not specify a COMMON name, BASIC supplies a default PSECT name of $BLANK on VAX/VMS systems and .$$$$. on PDP–11 systems. The following sections explain how to use static storage.

5.7.1 COMMON Statement

The COMMON statement defines a named area of storage (called a PSECT). Any BASIC subprogram can access the values in a COMMON by specifying a COMMON with the same name. The COMMON statement has the format:

```
{ COM    }
{ COMMON }  [ ( com-nam ) ] { [ data-type ] com-item },...
```

where:

com-nam Is the name of the COMMON block.

data-type Is a data-type keyword.

com-item Can be:

- A numeric variable
- A numeric array
- A fixed-length string variable
- An array of fixed-length strings
- A RECORD instance (VAX–11 BASIC only)
- A FILL item

The amount of storage reserved for a variable depends on its data type.

You can specify a length for string variables and string array elements that appear in a COMMON statement. If you do not specify a length, the default is 16. The format for specifying strings in a COMMON is:

```
COMMON (com-nam)  { str-vbl }  [= int-cnst]
                  { str-arr }
```

where:

int-cnst Is the length of the string variable or string array element.

For example:

```
10 COMMON (CODE) STRING EMP.CODE=2%, WAGE.CODE=3%, DEP.CODE=22%
```

This statement specifies 2 bytes for EMP.CODE, 3 bytes for WAGE.CODE, and 22 bytes for DEP.CODE.

In a single program module, multiple COMMONs with the same name allocate storage end-to-end in a single PSECT. That is, BASIC concatenates all COMMONs with the same name in the same program module, in the order they appear. For example:

```
100      COMMON (INTS) LONG CALL_COUNT, SUB1_COUNT, SUB2_COUNT
200      COMMON (INTS) LONG SUB3_COUNT, SUB4_COUNT
```

These two COMMONs allocate storage for five LONG integers in a single PSECT named INTS.

5.7.2 MAP Statement

The MAP statement, like the COMMON statement, creates a named area of static storage. However, if a program module contains multiple MAPs with the same name, the MAPs are overlaid on the same area of storage, rather than being concatenated.

When used with the MAP clause of the OPEN statement, the storage allocated by the MAP statement becomes the record buffer for that file. Variables in the MAP statement correspond to fields in the file's records.

See Chapter 4 to use the MAP statement to manipulate strings.

The MAP statement format is:

MAP (map-nam) { [data-type] map-item },...

where:

map-nam Is the name of the MAP block.

data-type Is a data-type keyword.

map-item Can be one of the following:

- A numeric variable
- A numeric array
- A fixed-length string variable
- An array of fixed-length strings
- A RECORD instance (VAX–11 BASIC only)
- A FILL item

5.7.2.1 Single MAPs

You associate a MAP with a record buffer by referencing the MAP in the OPEN statement.

Note

PDP–11 BASIC–PLUS–2 initializes variables named in MAP statements. However, VAX–11 BASIC does not.

The MAP statement must have a lower line number than any reference to MAP variables. For example:

```
10    ON ERROR GOTO 5000
20    DECLARE INTEGER CONSTANT EOF = 11

2000 MAP (PAYROL) STRING EMP_NAME, LONG WAGE_CLASS,      &
                        STRING SAL_REV_DATE, SINGLE TAX_YTD

2100 OPEN "PAYROL.DAT" FOR INPUT AS FILE #4%              &
                ,ORGANIZATION SEQUENTIAL                  &
                ,ACCESS READ                              &
                ,MAP PAYROL

2200 OPEN "PAYROL.NEW" FOR OUTPUT AS FILE #5%             &
                ,ORGANIZATION SEQUENTIAL                  &
                ,ACCESS WRITE                             &
                ,MAP PAYROL

3000 PRINT "PAYROLL VERIFICATION"


 GET_LOOP:
        WHILE 1% = 1%
         GET #4
         PRINT EMP_NAME, WAGE_CLASS%, SAL_REV_DATE, TAX_YTD
         PRINT "YOU CAN CHANGE:"
         PRINT "1,  EMPLOYEE NAME"
         PRINT "2,  WAGE CLASS"
         PRINT "3,  REVIEW DATE"
         PRINT "4,  TAX YEAR-TO-DATE"
         PRINT "5,  DONE"

 READ_LOOP:

          WHILE 1% = 1%

           INPUT "CHANGES?  ANSWER WITH YES OR NO" ; CHNG$
           IF CHNG$ = "NO" THEN ITERATE GET_LOOP
                     ELSE INPUT "NUMBER" ;NUMBER%
           END IF

           SELECT NUMBER%
           CASE 1
                INPUT "EMPLOYEE NAME"; EMP_NAME
           CASE 2
                INPUT "WAGE CLASS"; WAGE_CLASS
           CASE 3
                INPUT "REVIEW DATE";SAL_REV_DATE
           CASE 4
                INPUT "TAX YEAR-TO-DATE"; TAX_YTD
           CASE 5
                EXIT READ_LOOP
           CASE ELSE
                PRINT "Invalid response -- please try again"
           END SELECT
```

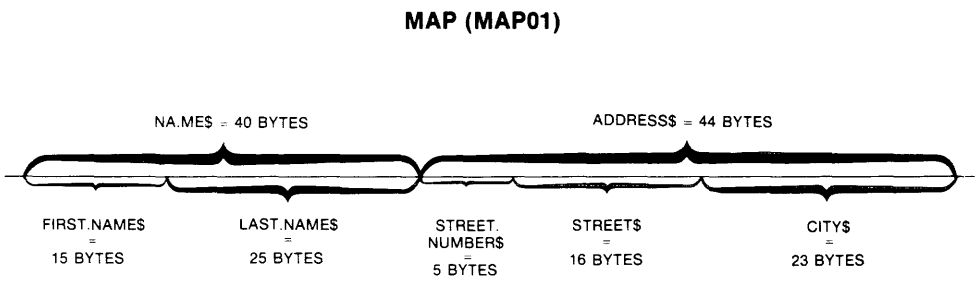
```
          NEXT
        PUT #5
       NEXT
5000    IF ERR = EOF
        THEN
                PRINT "End of file"
        ELSE
                ON ERROR GOTO 0
        END IF
32767 END
```

This program uses MAP variables to access fields in payroll records. Changes to MAP variables do not change the actual records in the file. To transfer the changed variables to the file, you must use the PUT statement (see Chapter 9).

5.7.2.2 Multiple MAPs

When a program contains more than one MAP with the same name, the storage allocated by these MAP statements is overlaid. This technique is useful for manipulating strings (as described in Chapter 4) and for accessing a record buffer in different ways. See Figure 5–2.

Figure 5–2: Multiple MAPs



When you use more than one MAP to access a record buffer, BASIC uses the size of the largest MAP to determine the size of the record. (The RECORDSIZE clause of the OPEN statement can override this MAP–defined recordsize. See Chapter 9.)

You can also use multiple MAPs to interpret numeric data in more than one way. For example:

```
100     MAP (BARRAY) BYTE ALPHABET(25)
110     MAP (BARRAY) STRING ABC = 26
120     FOR I% = 0% TO 25%
            ALPHABET(I%) = I% + 65%
        NEXT I%
130     PRINT ABC

RUNNH

ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

This example creates a MAP area named BARRAY. The first MAP statement allocates 26 bytes of storage in the form of an integer BYTE array. The second MAP statement defines this same storage as a 26–byte string named ABC. When the FOR–NEXT loop executes, it assigns values corresponding to the ASCII values for the uppercase letters A through Z.

5.7.3 FILL Items

FILL items reserve space in MAP and COMMON blocks and in record buffers accessed by MOVE statements. Thus, FILL items mask parts of the record buffer and let you: 1) skip over fields and 2) reserve space in or between data elements.

FILLs are available for all data types. Table 5–3 summarizes FILL formats and default allocations if no data type is specified.

Table 5–3: FILL Item Formats, Representations, and Default Allocations

FILL Format	Representation	Bytes Used
FILL	Floating-point	4, 8, or 16
FILL(n)	n floating-point elements	4n, 8n, or 16n
FILL%	Integer (BYTE, WORD, or LONG)	1, 2, or 4
FILL%(n)	n integer elements	1n, 2n, or 4n
FILL$	String	16
FILL$(n)	n string elements	16n
FILL$ = m	String	m
FILL$(n) = m	n string elements, m bytes each	m * n

Note

In the applicable formats of FILL, n represents a repeat count, not an array subscript. FILL(n), for example, represents n real elements, not n+1.

You can also use data-type keywords with FILL. If you do, the only valid form is FILL. The data-type and storage requirements are those of the last data type specified. For example:

```
100 MAP (QED) STRING A, FILL=24, LONG SSN, FILL, REAL SAL, FILL(5)
```

This MAP statement uses data-type keywords to reserve space for:

- A 16–character string variable A
- Twenty-four bytes of padding
- One LONG variable, SSN
- Four bytes of padding
- One REAL variable, SAL
- Space for five floating-point numbers. This requires 10, 20, or 80 bytes of padding, depending on the default size for floating-point numbers.

In VAX–11 BASIC, you can specify user-defined data types (RECORD names) for FILL items. For example:

```
100     RECORD X
          REAL Y1, Y2(10)
        END RECORD X
200     MAP (QED) X FILL
```

Line 100 defines a RECORD of data-type X. The MAP statement in line 200 contains a fill item of this data type, thus reserving space in the buffer for one RECORD of type X. See *BASIC on VAX/VMS Systems* for more information on the RECORD statement.

5.7.4 Using COMMON and MAP in Subprograms

The COMMON and MAP statements create a block of storage called a PSECT (program section). This COMMON or MAP storage block is accessible to any subprogram. Thus, a BASIC main and subprogram can share such an area by referencing the same COMMON or MAP name. For example:

Main Program

```
10      COMMON (A1) STRING A, B = 10, LONG C
```

Subprogram

```
10      COMMON (A1) STRING X, Z = 10, LONG Y
```

This COMMON block contains:

- A 16–character string field called A by the main program and X by the subprogram
- A 10–character string field called B by the main program and Z by the subprogram
- A 4–byte integer field called C by the main program and Y by the subprogram

If a subprogram defines a COMMON or MAP area with the same name as a COMMON or MAP in the main program, it overlays the COMMON or MAP defined in the main program.

It is important to note that multiple COMMON statements with the same name behave differently depending on whether these statements are in the same program module. If they are in the same program module, then the storage for each COMMON area is concatenated. However, if they are in different program units, then the COMMON areas overlay the same storage. For example:

```
100     COMMON (XYZ) STRING A = 32
200     COMMON (XYZ) STRING B = 32
```

Because these COMMON statements are in the same program module, they are concatenated in a single PSECT. Thus, the PSECT contains two 32–byte strings. In contrast:

Main Program

```
100     COMMON (XYZ) STRING A = 32
```

Subprogram

```
100     COMMON (XYZ) STRING B = 32
```

Because these COMMON statements are in different program modules, they overlay the same storage. Thus, the PSECT contains one 32–byte string, called A in the main program and B in the subprogram.

Although you can redefine the storage in a COMMON section when you access it from a subprogram, you should not do so. COMMON areas should contain exactly the same variables in all program modules. To make sure of this, you should use the %INCLUDE directive. For example:

COMMON.BAS (%INCLUDE file)

```
COMMON (SHARE) WORD EMP_NUM,                    &
               DECIMAL (8,0) SALARY,            &
               STRING WAGE_CLASS = 2
```

Main Program

```
100     %INCLUDE "COMMON.BAS"
```

Subprogram

```
200     %INCLUDE "COMMON.BAS"
```

Using the %INCLUDE directive to include the COMMON area definition means that there is much less chance of a typographical error. See Chapter 11 for more information on the %INCLUDE directive.

If you must redefine the variables in a PSECT, you should use the MAP statement. It is best to do this by using the %INCLUDE directive to create identical MAPs before redefining them. For example:

MAP.BAS (%INCLUDE file)

```
MAP (REDEF) STRING FULL_NAME = 40
```

Main Program

```
100     %INCLUDE "MAP.BAS"
```

Subprogram

```
200 %INCLUDE "MAP.BAS"
110 MAP (REDEF) STRING FIRST_NAME=15, MI=1, LAST_NAME=24
```

The MAP defined in MAP.BAS is included in both program modules as a 40–byte string. This MAP is redefined in the subprogram, allowing the subprogram to access parts of this string.

5.8 Dynamic Mapping

Dynamic mapping lets you redefine the position of variables in a MAP at run time. Dynamic mapping requires three BASIC statements:

- A MAP statement, which determines the total size of the storage area
- A MAP DYNAMIC statement, which names the variables whose positions can change at run time
- A REMAP statement, which specifies the new positions of the variables named in the MAP DYNAMIC statement

The MAP DYNAMIC statement is non-executable. Its format is:

MAP DYNAMIC (map-name) { [data-type] map-item } ,...

where:

map-name Is the storage area named in a previous MAP statement.

data-type Is any valid BASIC data-type keyword. In VAX–11 BASIC, data-type can also be a RECORD name.

map-item Can be an unsubscripted variable or an array.

Note that you cannot specify a string length in a MAP DYNAMIC statement. All string items have a length of zero until the program executes a REMAP statement.

The MAP DYNAMIC statement causes BASIC to create internal pointers to the variables and array elements. The MAP DYNAMIC statement does not affect the amount of storage allocated. Until your program executes the REMAP statement, the storage for each variable and each array element starts at the beginning of the MAP storage area.

The REMAP statement specifies the new positions of variables named in the MAP DYNAMIC statement. That is, it causes BASIC to change the internal pointers to the data. The format of REMAP is:

REMAP (map-name) [data-type] remap-item [= int-exp] ,...

where:

map-name Is the name of the MAP storage area.

data-type Is any valid BASIC data-type keyword. In VAX–11 BASIC, data-type can also be a RECORD name. In the REMAP statement, data-type can be specified only for FILL items.

remap-item Any variable, array, or array element named in the MAP DYNAMIC statement, or a FILL item.

int-exp Specifies the length of string variables, string array elements or string FILL items.

For example:

```
100    MAP (MAP01) STRING DUMMY = 116
       MAP DYNAMIC (MAP01) LONG A, STRING B, DOUBLE C(10)
       REMAP (MAP01) B = 24, A, C()
```

In this example, the MAP statement creates a storage area of 116 bytes and names it MAP01. The MAP DYNAMIC statement specifies that the position of variables A and B, and each of the elements of array C, can be dynamically redefined with the REMAP statement.

At the start of program execution, the storage for A is the first four bytes of MAP01, and the storage for each element of array C is the first eight bytes of MAP01. The storage for string variable B begins at the first byte of MAP01; however, string variables have a length of zero until the REMAP statement executes. The REMAP statement redefines these pointers so that:

- STRING variable B occupies the first 24 bytes of MAP01
- LONG variable A occupies the next 4 bytes of MAP01

- Array element C(0) occupies the next 8 bytes of MAP01, C(1) occupies the next 8 bytes, and so on. Array C uses a total of 88 bytes.

Because the REMAP statement is executable, it can redefine the pointer for a variable or array element more than once in a single REMAP statement. For example:

```
100     MAP (MAP01) STRING DUMMY = 116
        MAP DYNAMIC (MAP01) LONG A, STRING B, DOUBLE C(10)
        REMAP (MAP01) C(), C(0)
```

In this example, REMAP first causes array C to occupy the first 88 bytes of MAP01, then redefines C(0) to occupy the next 8 bytes in MAP01. Any reference to C(0) will affect bytes 89 through 96.

Note that DYNAMIC MAP variables are local to the program module in which they reside. Thus, REMAP only affects how that module views the buffer.

Chapter 6
Functions

Functions perform numeric or string operations on operands and return the result to your program. BASIC has built-in functions that perform numeric, string, conversion, date, and time operations. This chapter demonstrates the use of a selected group of built-in functions. Chapter 4 also describes some built-in string functions. For a complete list of BASIC built-in functions, see the *BASIC Language Reference Manual*.

This chapter also describes user-defined functions. BASIC lets you define your own functions in two ways:

- With the DEF statement
- As separately compiled subprograms (external functions)

DEF function definitions are local to a program module, while external functions can be accessed by any program module. You create local functions with the DEF statement and optionally declare them with the DECLARE statement. You create external functions with the FUNCTION statement and declare them with the EXTERNAL statement.

Once a function has been created and declared, you call the function just as you would a built-in function.

Note

For compatibility with BASIC–PLUS, BASIC–PLUS–2 and VAX–11 BASIC also support the DEF* function definition statement. DEF* statements are not recommended for new program development.

6.1 BASIC Built-in Functions

The functions described in this section let you perform sophisticated manipulation of string and numeric data. A function performs a numeric or string operation on a specified argument and returns the result to your program. There are numeric, string, conversion, date, and time functions in the BASIC library.

BASIC also provides algebraic, exponential, trigonometric, and randomizing mathematic functions. BASIC converts numeric arguments to the proper data type when you invoke a built-in function. Thus, if you supply an integer argument to a function that expects a floating-point number, BASIC converts the argument to floating-point. Note that floating-point arguments to integer functions are truncated, not rounded.

6.1.1 Using Numeric Functions

Numeric functions generally return a result of the same data type as the function's parameter. Of course, this is not true for functions that are designed to return a result of a particular data type. For example, if you pass a DOUBLE argument to any of the trigonometric functions, they return a DOUBLE result.

On the other hand, the ABS function returns a floating-point number equivalent to the absolute value of the argument. Thus, ABS always returns a result of the default floating-point data type.

Also, if the format of a BASIC built-in function specifies an argument of a particular data type, this implies that BASIC converts the actual argument you supply to the specified data type.

6.1.1.1 ABS Function

The ABS function returns a floating-point number that equals the absolute value of a specified numeric expression. Its format is:

ABS(num-exp)

where:

num-exp Is any numeric expression.

For example:

```
10 READ A,B,C,D,E
20 DATA 10,-35.3,600,-9,0
30 PRINT ABS(A); ABS(B); ABS(C); ABS(D); ABS(E)
40 END

RUNNH

 10   35.3   600   9   0
```

ABS always returns a number of the default floating-point data type.

6.1.1.2 SGN Function

The SGN function determines whether a number is positive, negative, or zero. Its format is:

SGN(num-exp)

where:

num-exp Is any numeric expression.

If the expression is positive, SGN returns a positive 1; if negative, a negative 1. If the expression equals zero, SGN returns zero. For example:

```
10 A = -7.32
20 B = .44
30 C = 0
40 PRINT "A = ";A ,"B = ";B ,"C = ";C
50 PRINT "SGN(A)="; SGN(A),
60 PRINT "SGN(B)="; SGN(B),
70 PRINT "SGN(C)="; SGN(C)
80 END

RUNNH

A=-7.32             B= .44          C= 0
SGN(A)=-1           SGN(B)= 1       SGN(C)= 0
```

The SGN function always returns a number of the default integer type.

6.1.1.3 INT and FIX Functions

The INT function returns the floating-point value of the largest integer less than or equal to a specified expression. Its format is:

INT(exp)

where:

exp Is any numeric expression.

INT always returns a number of the default floating-point type.

The FIX function truncates the value of a floating-point number at the decimal point. Its format is:

FIX(num-exp)

where:

num-exp Is any numeric expression.

FIX always returns a number of the default floating-point type.

This is an example of INT:

```
10 PRINT INT(23.553)
20 PRINT INT(3.1)
30 PRINT INT(-45.3)
40 PRINT INT(-11)
50 END

RUNNH

 23
 3
-46
-11
```

This is an example of FIX:

```
10 PRINT FIX(23.553)
20 PRINT FIX(3.1)
30 PRINT FIX(-45.3)
40 PRINT FIX(-11)
50 END

RUNNH

 23
 3
-45
-11
```

Note that the value returned by FIX(–45.3) differs from the value returned by INT(–45.3).

6.1.1.4 SIN, COS, and TAN Functions

The SIN, COS, and TAN functions return the sine, cosine, and tangents of an angle. Their formats are:

SIN(num-exp)

COS(num-exp)

TAN(num-exp)

where:

num-exp Is an angle in radians.

If you supply a floating-point argument to the SIN, COS, and TAN functions, they return a number of the same floating-point type. However, if you supply an integer argument, they convert the argument to the default floating-point data type and return a floating-point number of that type.

This program accepts an angle in degrees, converts the angle to radians, and prints the angle's sine, cosine, and tangent:

```
10            REM - CONVERT ANGLE (X) TO RADIANS, AND
20            REM - FIND SIN, COS AND TAN
30            PRINT "DEGREES", "RADIANS", "SINE", "COSINE","TANGENT"
40            FOR I% = 0% TO 5%
                  READ X
                  LET Y = X * 2 * PI / 360
                  PRINT
                  PRINT X ,Y ,SIN(Y) ,COS(Y) ,TAN(Y)
              NEXT I%
50            DATA 0,10,20,30,360,45
60            END

RUNNH

     DEGREES     RADIANS     SINE        COSINE      TANGENT

      0          0           0           1           0

      10         .174533     .173648     .984808     .167327

      20         .349066     .34202      .939693     .36397
```

```
30          .523599     .5          .866025     .57735

360         6.28319     0           1           0

45          .785398     .707107     .707107     1
```

Note

As an angle approaches 90 degrees (PI/2 radians), 270 degrees (3*PI/2 radians), 450 degrees (5*PI/2 radians) and so on, the tangent of that angle approaches infinity. If your program tries to find the tangent of such an angle, BASIC signals "Division by 0" (ERR=61).

6.1.1.5 LOG10 Function

A logarithm is the exponent of another number (called a base). Common logarithms use the base 10. The common logarithm of a number N, therefore, is the power to which 10 must be raised to equal N. For example, the common logarithm of 100 is 2, because 10 raised to the power 2 equals 100.

The LOG10 function returns a number's common logarithm. Its format is:

LOG10(num-exp)

where:

num-exp Is any numeric expression.

For example:

```
10      FOR I% = 10% TO 100% STEP 10%
            PRINT LOG10(I%)
        NEXT I%
40 END

RUNNH

 1
 1.30103
 1.47712
 1.60206
 1.69897
 1.77815
 1.8451
 1.90309
 1.95424
 2
```

If you supply a floating-point argument to LOG10, the function returns a floating-point number of the same data type. However, if you supply an integer argument, LOG10 converts it to the default floating-point data type and returns a value of that type.

6.1.1.6 LOG Function

Natural logarithms are exponents of the base "e", where "e" is a mathematical constant approximately equal to 2.71828. That is, the natural logarithm of a number N is the power to which "e" must be raised to equal N: the natural logarithm of 100 is 4.60517, because "e" raised to the power 4.60517 equals 100.

The LOG function returns the natural (base "e") logarithm of a number. Its format is:

LOG(num-exp)

where:

num-exp Is any numeric expression.

For example:

```
10 READ A, B
20 DATA 2.71828, 100
30 PRINT "THE NATURAL LOG OF";A;"IS";LOG(A)
40 PRINT "THE NATURAL LOG OF";B;"IS";LOG(B)
50 END

RUNNH

THE NATURAL LOG OF 2.71828 IS 1
THE NATURAL LOG OF 100 IS 4.60517
```

If you supply a floating-point argument to LOG, the function returns a floating-point number of the same data type. However, if you supply an integer argument, LOG converts it to the default floating-point data type and returns a value of that type.

6.1.1.7 EXP Function

The EXP function returns the value of "e" raised to a specified power. Its format is:

EXP(exp)

where:

exp Is the power to which BASIC raises "e".

For example:

```
10 READ A,B
20 DATA 1,2
30 PRINT '"e" RAISED TO THE POWER'; A; "EQUALS"; EXP(A)
40 PRINT '"e" RAISED TO THE POWER'; B; "EQUALS"; EXP(B)
50 END

RUNNH

"e" RAISED TO THE POWER 1 EQUALS 2.71828
"e" RAISED TO THE POWER 2 EQUALS 7.38906
```

If you supply a floating-point argument to EXP, the function returns a floating-point number of the same data type. However, if you supply an integer argument, EXP converts it to the default floating-point data type and returns a value of that type.

6.1.1.8 ATN Function

The ATN function returns the angle, in radians, of a specified tangent. Its format is:

ATN(tangent-value)

The ATN function returns a value between PI/2 and –PI/2, inclusive. For example:

```
10 INPUT "TANGENT VALUE"; T
20 ATN.OF.T = ATN(T)
30 PRINT "SMALLEST ANGLE WITH THAT TANGENT";
40 PRINT "IS"; ATN.OF.T; "RADIANS"
50 ANG.IN.DEG = ATN.OF.T / (PI / 180)
60 PRINT "AND"; ANG.IN.DEG; "DEGREES"
70 END
```

If you supply a floating-point argument to ATN, the function returns a floating-point number of the same data type. However, if you supply an integer argument, ATN converts it to the default floating-point data type and returns a value of that type.

6.1.1.9 RND Function

The RND function returns a number greater than or equal to zero and less than one. Its format is:

RND

The RND function always returns a floating-point number of the default floating-point data type. The RND function generates seemingly unrelated numbers. However, given the same starting conditions, a computer always gives the same results. Each time you execute a program with the RND function, you receive the same results. For example:

```
10 PRINT RND,RND,RND,RND
20 END

RUNNH

 .76308        .179978        .902878        .88984

RUNNH

 .76308        .179978        .902878        .88984
```

With the RANDOMIZE statement, you can change the RND function's starting condition and generate truly random numbers. To do this, place a RANDOMIZE statement before the line invoking the RND function. Note that the RANDOMIZE statement should be used only once in a program. With the RANDOMIZE statement, each invocation of RND returns a new and unpredictable number. For example:

```
10 RANDOMIZE
20 PRINT RND,RND,RND,RND
30 END

RUNNH

  .403732        .34971         .15302         .92462

RUNNH

  .404165        .272398        .261667        .10209
```

The RND function can generate a series of random numbers over any open range. To produce random numbers in the open range A to B, use the formula:

(B–A)*RND + A

This program produces 10 numbers in the open range 4 to 6:

```
10        FOR I% = 1% TO 10%
              PRINT (6%-4%) * RND + 4
          NEXT I%
40        END

RUNNH

 5.52616
 4.35996
 5.80576
 5.77968
 4.77402
 4.95189
 5.76439
 4.37156
 5.2776
 4.53843
```

6.1.1.10 SQR Function

The SQR function returns a positive number's square root. Its format is:

SQR(num-exp)

For example:

```
10        FOR I% = 1% TO 2%
              READ A
              PRINT SQR(A)
          NEXT I%
          DATA 10,4
30        END

RUNNH

 3.16228
 2
```

If the argument to the SQR function is less than zero, BASIC signals "Imaginary square root" (ERR = 54).

The SQR function always returns a number of the default floating-point data type.

6.2 Using Data Conversion Functions

BASIC provides the means to:

- Convert a 1–character string to the character's ASCII value and vice versa (ASCII and CHR$ functions)
- Translate strings from one data format to another, for example, EBCDIC to ASCII (XLATE function)

The following sections describe these functions.

6.2.1 ASCII Function

The ASCII function returns the numeric ASCII value of a string's first character. Its format is:

ASCII(str-exp)

The ASCII function returns an integer value between 0 and 255, inclusive. For example:

```
10 TEST.STRING$ = "BAT"
20 PRINT ASCII(TEST.STRING$)
30 END

RUNNH

 66
```

Line 20 prints 66 because this is the ASCII value of an uppercase B, the first character in the string.

The ASCII value of a null string is zero.

6.2.2 CHR$ Function

The CHR$ function returns the character whose ASCII value you supply. Its format is:

CHR$(num-exp)

where:

num-exp Is a numeric expression. If num-exp is less than zero or greater than 255, it is treated modulo 256; thus, CHR$(325) is equivalent to CHR$(69) and CHR$(–1) is equivalent to CHR$(255).

For example:

```
10 PRINT "THIS PROGRAM FINDS THE CHARACTER WHOSE"
20 PRINT "VALUE (MODULO 256) YOU TYPE"
30 INPUT VALUE%
40 PRINT CHR$(VALUE%)
50 END

RUNNH

THIS PROGRAM FINDS THE CHARACTER WHOSE
VALUE (MODULO 256) YOU TYPE
 ? 69
E
```

6.2.3 XLATE Function

The XLATE function translates one string to another by referencing a table you provide. Its format is:

XLATE(str-exp1, str-exp2)

where:

str-exp1 Is the input string.

str-exp2 Is the table string.

The table string can contain up to 256 ASCII characters; the position of each character in the table string corresponds to an ASCII value. Because zero is a valid ASCII value (null), the first position in the table string is position zero.

XLATE scans the input string character by character, from left to right. It finds the ASCII value n of the input character and extracts the character it finds at position n in the table string. XLATE then appends the table string character to the returned string and goes on to the next character in the input string. For example, this program uses XLATE to convert uppercase letters to lowercase:

```
10 A$ = "abcdefghijklmnopqrstuvwxyz"
20 TABLE$ = STRING$(65%,0%) + A$
30 LINPUT "TYPE A STRING OF UPPERCASE LETTERS"; SOURCE$
40 PRINT XLATE(SOURCE$, TABLE$)
50 END

RUNNH

TYPE A STRING OF UPPERCASE LETTERS? XYZZY
xyzzy
```

The table string contains 65 nulls and the 26 lowercase letters. In line 40, XLATE takes the ASCII value of X (88) and extracts the character at position 88 in the table string (the first character in the table string is position zero). This character is a lowercase x. BASIC appends it to the output string, and proceeds in this way until there are no more characters in the input string.

These restrictions apply to XLATE:

- XLATE does not translate nulls. Therefore, the output string may be smaller than the input string.
- If the character at position n in the table string is a null, XLATE does not translate the character whose ASCII value is n.
- If the ASCII value of the input character is outside the range of positions in the table string, XLATE does not translate that input character.

6.2.4 Using Numeric String Functions

Numeric strings are numbers represented by ASCII characters. A numeric string consists of an optional sign, a string of digits, and an optional decimal point. You can use E notation in the numeric string for floating-point constants.

The following sections describe functions for printing, converting, and evaluating numeric strings.

6.2.4.1 FORMAT$ Function

The FORMAT$ function converts a numeric value to a string. The output string is formatted according to a string you provide. Its format is:

FORMAT$(exp, format-string)

where:

exp	Is any string or numeric expression.
format-string	Is a string expression containing at least one PRINT USING format field.

The formatting rules are the same as those for printing numbers with PRINT USING. See Chapter 8 for more information on formatting rules. For example:

```
10 A = 5
20 B$ = "##.##"
30 Z$ = FORMAT$(A, B$)
40 PRINT Z$
50 END

RUNNH

 5.00
```

6.2.4.2 NUM$ and NUM1$ Functions

The NUM$ function evaluates a numeric expression and returns a string of characters formatted as the PRINT statement would format it. The NUM$ format is:

NUM$(num-exp)

The returned numeric string is preceded by one space for positive numbers and by a minus sign for negative numbers. The numeric string is always followed by a space. For example:

```
10 PRINT NUM$(7465097802134)
20 PRINT NUM$(-50)
30 END

RUNNH

 .74651E+13
-50
```

The NUM1$ function translates a number into a string of numeric characters. NUM1$ does not return leading or trailing spaces or E format. The NUM1$ format is:

NUM1$(num-exp)

For example:

```
10 PRINT NUM1$(PI)
20 PRINT NUM1$(97.5 * 30456.23 + 30385.1)
30 PRINT NUM1$(1E-38)
40 END

RUNNH

3.14159
2999870
.00000000000000000000000000000000000001
```

NUM1$ returns up to 6 digits of accuracy for single-precision real numbers, up to 16 digits of accuracy for double-precision numbers, and up to 10 digits of accuracy for LONG integers. On VAX/VMS systems, NUM1$ returns up to 15 digits of accuracy for GFLOAT numbers and up to 33 digits of accuracy for HFLOAT numbers.

This example shows the difference between NUM$ and NUM1$:

```
10 A$ = NUM$(1000000)
20 B$ = NUM1$(1000000)
30 PRINT LEN(A$); "/"; A$; "/"
40 PRINT LEN(B$); "/"; B$; "/"
50 END

RUNNH

 8 / .1E+07 /
 7 /1000000/
```

Note that A$ has a leading and trailing space.

6.2.4.3 STR$ Function

The STR$ function converts a numeric expression to a string of ASCII digits. STR$ is identical to NUM$ except that the returned string has no leading or trailing spaces. Its format is:

STR$(num-exp)

For example:

```
10 A = 10
20 B$ = STR$(A)
30 PRINT B$
40 PRINT A
50 END

RUNNH

10
 10
```

In line 20, B$ contains the ASCII characters "1" and "0". BASIC prints B$ with no leading space because it is a string, but it prints A with a leading and trailing space because it is a number.

6.2.4.4 VAL% and VAL Functions

The VAL% function returns the integer value of a numeric string. Its format is:

VAL%(num-str-exp)

where:

num-str-exp Must be an integer's string representation. It can contain:

- The ASCII digits 0 through 9
- The symbols "+" and "–"

The VAL function returns the floating-point value of a numeric string. Its format is:

VAL(num-str-exp)

where:

num-str-exp Must be a number's string representation. It can contain:

- The ASCII digits 0 through 9
- The symbols "+", "–", and "."
- An uppercase E

VAL returns a number of the default floating-point data type. BASIC signals "Illegal number" (ERR = 52) if the argument is outside the range of the default floating-point data type.

This is an example of VAL and VAL%:

```
10 A = VAL("922")
20 B$ = "100"
30 C% = VAL%(B$)
40 PRINT A
40 PRINT C%
60 END

RUNNH

 922
 100
```

6.2.5 Using String Arithmetic Functions

String arithmetic functions process numeric strings as arithmetic operands. This lets you add (SUM$), subtract (DIF$), multiply (PROD$), or divide (QUO$) numeric strings or express them at a specified level of precision (PLACE$).

String arithmetic offers greater precision than floating-point arithmetic or longword integers and eliminates the need for scaling. However, string arithmetic executes much more slowly than integer or floating-point operations.

The operands for the functions can be numeric strings representing any integer or floating-point value (E notation is not valid). Table 6–1 shows the string arithmetic functions and their formats, and gives brief descriptions of what they do. Later sections give more detailed descriptions of each string function.

Table 6–1: String Arithmetic Functions

Function	Format	Description
SUM$	SUM$(A$,B$)	B$ is added to A$.
DIF$	DIF$(A$,B$)	B$ is subtracted from A$.
PROD$	PROD$(A$,B$,P%)	A$ is multiplied by B$. The product is expressed with precision P%.
QUO$	QUO$(A$,B$,P%)	A$ is divided by B$. The quotient is expressed with precision P%.
PLACE$	PLACE$(A$,P%)	A$ is expressed with precision P%.

SUM$ and DIF$ take the precision of the more precise argument in the function, unless padded zeros generate that precision. SUM$ and DIF$ omit trailing zeros to the right of the decimal point.

String arithmetic computations permit 56 significant digits. The functions QUO$, PLACE$, and PROD$, however, permit up to 60 significant digits. Table 6–2 shows how BASIC determines the precision permitted by each function and if that precision is implicit or explicit.

Table 6–2: Precision of String Arithmetic Functions

Function	How Determined	How Stated
SUM$	Precision of argument	Implicitly
DIF$	Precision of argument	Implicitly
PROD$	Value of argument	Explicitly
QUO$	Value of argument	Explicitly
PLACE$	Value of argument	Explicitly

The size and precision of results returned by the SUM$ and DIF$ functions depend on the size and precision of the arguments involved:

- The sum or difference of two integers takes the precision of the larger integer.
- The sum or difference of two decimal fractions takes the precision of the more precise fraction.
- The sum or difference of two real numbers takes precision as follows:
  - The sum or difference of the integer parts takes the precision of the larger part.
  - The sum or difference of the decimal fraction parts takes the precision of the more precise part.
- BASIC truncates trailing zeros.

In the PLACE$, PROD$, and QUO$ functions, the value of the integer expression explicitly determines numeric precision. That is, the integer expression determines the point at which the number is rounded or truncated.

If the integer expression is between –5000 and 5000, rounding occurs according to the following rules:

- For positive integer expressions, rounding occurs to the right of the decimal place. For example, if the integer expression is 1, rounding occurs one digit to the right of the decimal place (the number is rounded to the nearest tenth). If 2, rounding occurs two digits to the right of the decimal place (the number is rounded to the nearest hundredth), and so on.
- For zero, BASIC rounds to the nearest unit.
- For negative integer expressions, rounding occurs to the left of the decimal place. For example, if the integer expression is –1, rounding occurs one place to the left of the decimal point. In this case, BASIC moves the decimal point one place to the left, then rounds to units. If the integer expression 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 units.

Note that when rounding numeric strings, BASIC returns only part of the number. See Section 6.2.5.1 for an example of this behavior.

If the integer expression is between 5001 and 15000, truncation occurs:

- If the integer expression is 10000, BASIC truncates the number at the decimal point.
- If the integer expression is greater than 10000 (10000 plus n) BASIC truncates the numeric string n places to the right of the decimal point. For example, if the integer expression is 10001 (10000 plus 1), BASIC truncates the number starting one place to the right of the decimal point. If 10002 (10000 plus 2), BASIC truncates the number starting two places to the right of the decimal point, and so on.
- If the integer expression is less than 10000 (10000 minus n) BASIC truncates the numeric string n places to the left of the decimal point. For example, if the integer expression is 9999 (10000 minus 1), BASIC truncates the number starting one place to the left of the decimal point. If 9998 (10000 minus 2), BASIC truncates starting two places to the left of the decimal point, and so on.

For examples of this rounding and truncation behavior, see the following explanation of the PLACE$ function.

6.2.5.1 PLACE$ Function

The PLACE$ function returns a numeric string, truncated or rounded according to an integer argument you supply. Its format is:

```
PLACE$(num-str, int-exp)
```

For example:

```
10              NUMBER$ = "123456.654321"
20              FOR I% = -5% TO 5%
                                          PRINT PLACE$(NUMBER$, I%)
                NEXT I%
30              PRINT
40              FOR I% = 9995 TO 10005
                                          PRINT PLACE$(NUMBER$, I%)
                NEXT I%

RUNNH

1
12
123
1235
12346
123457
123456.7
123456.65
123456.654
123456.6543
123456.65432

1
12
123
1234
12345
123456
123456.6
123456.65
123456.654
123456.6543
123456.65432
```

6.2.5.2 SUM$ Function

The SUM$ function returns the sum of two numeric strings. Its format is:

SUM$(num-str1, num-str2)

For example:

```
10 A$ = "56"
20 SIGMA$ = SUM$(A$, "44")
30 PRINT SIGMA$
40 END

RUNNH

100
```

6.2.5.3 DIF$ Function

The DIF$ returns the arithmetic difference of two numeric strings. Its format is:

DIF$(num-str1, num-str2)

The first numeric string is the minuend and the second is the subtrahend. For example:

```
10 A$ = "99999"
20 ANSWER$ = DIF$(A$, "11111")
30 PRINT ANSWER$
40 END

RUNNH

88888
```

6.2.5.4 PROD$ Function

The PROD$ function returns the product of two numeric strings. Its format is:

PROD$(num-str1, num-str2, int-exp)

The returned string's precision depends on the value of the integer expression. (See Section 6.2.5 for allowable values of int-exp.) For example:

```
10 A$ = "-4.333"
20 B$ = "7.23326"
30 S.PRODUCT$ = PROD$(A$, B$, 10005%)
40 PRINT S.PRODUCT$
50 END

RUNNH

-31.34171
```

6.2.5.5 QUO$ Function

The QUO$ function returns the quotient of two numeric strings. Its format is:

QUO$(num-str1, num-str2, int-exp)

where:

num-str1 Is the numerator.

num-str2 Is the denominator.

int-exp Determines rounding and truncation. (See Section 6.2.5 for allowable values of int-exp.)

For example:

```
10 NUMER$ = "32767"
20 DENOM$ = "65535"
30 PRECISION% = 10004%
40 QUOTIENT$ = QUO$(NUMER$, DENOM$, PRECISION%)
50 PRINT QUOTIENT$
60 END

RUNNH

.4999
```

6.2.6 Using Date and Time Functions

BASIC supplies functions to return the date and time in numeric or string format. The following sections discuss these functions.

On VAX/VMS systems you can use system services and Run-Time Library routines for more sophisticated date and time functions. See the *VAX/VMS System Services Reference Manual* and the *VAX/VMS Run-Time Library Reference Manual*.

6.2.6.1 DATE$ Function

The DATE$ function returns a string containing a day, month, and year in the form: DD–MMM–YY. (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.) The format of the DATE$ function is:

DATE$(num-exp)

The numeric expression's thousands digit represents the number of years since 1970. The remaining digits specify the day of that year. If the numeric expression is zero, DATE$ returns the current date. For example:

```
10 PRINT DATE$(0)
20 PRINT DATE$(126)
30 PRINT DATE$(6168)
50 END

RUNNH

16-May-82
06-May-70
16-Jun-76
```

If you supply an invalid date (for example, day 370 of 1973), results are undefined.

6.2.6.2 TIME$ Function

The TIME$ function returns a string displaying the time of day in the form HH:MM AM or HH:MM PM. Its format is:

TIME$(num-exp)

TIME$ returns the time of day at a specified number of minutes before midnight. If you specify zero in the numeric expression, TIME$ returns the current time of day. For example:

```
10 PRINT TIME$(0)
20 PRINT TIME$(1)
30 PRINT TIME$(1440)
40 PRINT TIME$(721)
50 END

RUNNH

01:53 PM
11:59 PM
12:00 AM
11:59 AM
```

6.2.6.3 TIME Function

The TIME function requests time and usage information from the operating system and returns it to your program. Because of this, TIME works differently on RSTS/E, RSX–11M/M–PLUS, and VAX/VMS systems. On VAX/VMS systems, its format is:

TIME(int-exp)

Where int-exp is a integer expression. If int-exp is:

0 TIME returns the number of seconds elapsed since midnight.

1 TIME returns the current job's CPU time in tenths of a second.

2 TIME returns the current job's connect time in seconds.

3 TIME returns a zero.

4 TIME returns a zero.

All other arguments to TIME are undefined and cause VAX–11 BASIC to signal "Not implemented" (ERR = 250).

On RSX–11M/M–PLUS systems, TIME accepts an argument of zero only. TIME(0%) returns a floating-point number that is the number of seconds that have elapsed since midnight.

On RSTS/E systems, the format is:

TIME(int-exp)

Where int-exp is an integer expression. If int-exp is:

0 TIME returns the number of seconds elapsed since midnight.

1 TIME returns the current job's CPU time in tenths of a second.

2 TIME returns the current job's connect time in seconds.

3 TIME returns the number of kilo-core ticks (KCTs) that your job used. See the *RSTS/E System User's Guide* for an explanation of KCTs.

4 TIME returns the device time for the job in minutes.

This example prints the number of seconds elapsed since midnight:

```
10 PRINT TIME(0)
20 END

RUNNH

 50755
```

6.2.7 Using Terminal Control Functions

Terminal control functions let you:

- Enable and disable CTRL/C trapping
- Enable and disable terminal echoing
- Remove the effect of a CTRL/O
- Tab to a specified character position

6.2.7.1 CTRLC and RCTRLC Functions

The CTRLC function enables CTRL/C trapping. The RCTRLC function disables CTRL/C trapping. When CTRL/C trapping is enabled, control is transferred to the program's error handler when a CTRL/C is detected at the controlling terminal. The format for CTRLC is:

int-vbl = CTRLC

The format for RCTRLC is:

int-vbl = RCTRLC

CTRL/C trapping is asynchronous. The trap can occur in the middle of an executing statement, and a statement so interrupted leaves variables in an undefined state. For example, the statement A$ = "ABC", if interrupted by CTRL/C, could leave the variable A$ partially set to "ABC" and partially left with its old contents. Therefore, you should use the CTRLC function only when doing a final cleanup before exiting a program. For example:

```
100 ON ERROR GOTO 19000
200 Y% = CTRLC
      !.
      !.
      !.
19000 REM ERROR HANDLER
19100 IF ERR = 28% THEN GOTO 20000
      !.
      !.
      !.
20000 REM CONTROL C--CLEANUP AND EXIT
20100 RESUME 20200
20200 PRINT #1, "ABORT"
20300 CLOSE #1
20400 END
```

If you type a CTRL/C to this program when CTRL/C trapping is enabled, an "ABORT" message prints to the file open on channel #1. This lets you know that the program did not end correctly.

6.2.7.2 ECHO and NOECHO Functions

The NOECHO function disables echoing on a specified channel. Echoing is the process by which characters typed at the terminal keyboard appear on the terminal screen or paper. The format of NOECHO is:

NOECHO(chnl-exp)

If you specify channel zero (your terminal) the characters typed on the keyboard are still accepted as input; however, they do not appear on the screen.

The ECHO function enables echoing on a specified channel and cancels the effect of the NOECHO function on that channel. Its format is:

ECHO(chnl-exp)

If you do not use these functions, ECHO is the default. This program shows a password routine in which the password does not echo:

```
10 Y% = NOECHO(0%)
20 INPUT "PASSWORD"; PWORD$
30 IF PWORD$=="PLUGH" THEN PRINT "THAT IS CORRECT"
40 Y% = ECHO(0%)
50 END
```

6.2.7.3 TAB Function

You use the TAB function with the PRINT statement. TAB moves the cursor or print mechanism to a specified column. Its format is:

PRINT TAB(int-exp)

where:

int-exp Specifies the column number of the cursor or print mechanism. The first column position on your terminal is position zero.

The TAB function can move the cursor to the right only. If the cursor is to the right of the specified column, TAB has no effect. For example:

```
10 PRINT "TEST"; TAB(2%); "TEST"
20 END

RUNNH

TESTTEST
```

Because the cursor is at position five when BASIC executes TAB(2%), it has no effect.

When using the TAB function, you should not use commas to separate PRINT elements. If you do, BASIC moves the cursor to the next print zone before executing the TAB. Instead, use semicolons. For example:

```
10 PRINT "NAME"; TAB(15%); "ADDRESS"; TAB(30%); "PHONE NO."
20 END

RUNNH

NAME            ADDRESS         PHONE NO.
```

6.3 The DEF Statement

The DEF statement lets you create your own single-line or multi-line functions.

In the traditional BASIC usage, a function name consists of:

- The letters FN
- 1 to 28 letters, digits, underscores, or periods
- An optional percent sign or dollar sign

Integer function names must end with a percent sign and string function names must end with a dollar sign. Therefore, the function name can have up to 31 characters. If the function name ends with neither a percent sign nor a dollar sign, the function returns a real number.

You can still create user-defined functions using these function names. However, DIGITAL recommends that you use explicit data typing when defining functions for new program development. Refer to Section 6.3.2 for an example of an explicitly declared function. Note that the function name must start with FN only if the function is not explicitly declared and a function reference lexically precedes the function definition.

DEF functions can be either single-line or multi-line. Whether you use the single-line or multi-line format for function definitions depends on the complexity of the function you create. In general, multi-line DEFs are for more complex functions than single-line DEFs. However, the important distinction between single- and multi-line DEFs is that multi-line DEFs can be invoked recursively, and single-line DEFs cannot.

If you want to pass values to a function, the function definition requires a formal parameter list. These formal parameters are the variables used to calculate the value returned by the function. When you invoke a function, you supply an actual parameter list; the values in the actual parameter list are copied into the formal parameter at this time. DEF functions allow up to eight formal parameters. You can specify variables, constants, or array elements as formal parameters, but you cannot specify an entire array as a parameter to a DEF function.

6.3.1 Single-Line DEFs

In a single-line DEF, the function name, the formal parameter list and the defining expression all appear on the same line. The format of a single-line DEF is:

```
DEF [data-type] func-nam [(formal-list)] = exp
```

where:

data-type Is any valid BASIC data-type keyword or a user-defined RECORD name (VAX–11 BASIC only).

func-nam Is the function name.

formal-list Is a list of zero to eight arguments to the function. These parameters are variables local to the function definition. Each formal parameter can be preceded by a data-type keyword.

exp Is the defining expression. This expression specifies the calculations the function performs.

For example:

```
100     DEF FNRATIO (NUMER, DENOMIN) = NUMER / DENOMIN
200     PRINT FNRATIO(5.6, 7.8)
300     END
```

Line 100 contains the function name; FNRATIO, the formal parameter list containing two variables, NUMER and DENOMIN; and the defining expression, NUMER / DENOMIN. The defining expression specifies the operations the function performs. In line 200, the function is invoked with two actual parameters: 5.6 and 7.8. These values are assigned to the corresponding formal parameters. The actual parameters you supply must agree in number and data type with those in the formal parameter list; that is, you must supply numeric values for numeric variables, and string values for string variables.

In the previous example, when the function is invoked in line 200, BASIC:

- Copies the values 5.6 and 7.8 into the formal parameters NUMER and DENOMIN
- Evaluates the expression to the right of the equal sign
- Returns the value to the statement that invoked the function (line 200)

The PRINT statement then prints the returned value.

The defining expression for a single-line function definition can contain any constant, variable, BASIC Library function, or user-defined function, except the function being defined. The following are valid function definitions:

```
10 DEF FN.A(X) = X^2 + 3 * X + 4
20 DEF FN.B(X) = FN.A(X) / 2 + FN.A(X)
30 DEF FN.C(X) = SQR(X+4) + 1
40 DEF CUBE(X) = X ^ 3
```

Note that the name of the function defined on line 40 does not begin with FN. This is valid as long as no reference to the function lexically precedes the function definition.

You can also define a function that has no formal parameters. For example:

```
1000    DEF FNDAY_NUMBER% = VAL% (SEG$(DATE$(0%), 1%, 2%))
```

This function definition uses three BASIC built-in functions to return an integer corresponding to the day of the month. DATE$(0) returns a date string in the form dd-Mmm-yy. The SEG$ function strips out of this value the characters starting at character position one up to and including the character at position two (the day number). The VAL% function converts this resulting numeric string to an integer. Thus, FNDAY_NUMBER% returns the day of the month as an integer.

6.3.2 Multi-Line DEFs

The DEF statement can also define multi-line functions. Multi-line DEFs are useful for expressing complicated functions. The format is:

```
DEF [data-type] func-nam [(formal-parameter-list)]
    statement(s)
END DEF
```

Note that multi-line DEFs do not have the equal sign and defining expression on the first line. Instead, this expression appears in the function block, assigned to the function name.

Note

If a multi-line DEF contains DATA statements, they are global to the program.

Multi-line function definitions can contain any constant, variable, BASIC Library function, or user-defined function. In VAX–11 BASIC, the function definition can contain a reference to the function you are defining. This means that in VAX–11 BASIC, a multi-line DEF can be recursive, or can invoke itself.

You signal the end of the function definition with the END DEF statement. Its format is:

```
END DEF
```

FNEND is a synonym for END DEF, but END DEF is preferred.

The EXIT DEF statement lets you exit from a user-defined function. It is equivalent to an unconditional transfer to the END DEF statement. Its format is:

```
EXIT DEF
```

FNEXIT is a synonym for EXIT DEF but EXIT DEF is preferred. This is an example of a multi-line DEF:

```
10      DEF FN.DISCONT(A)
            IF A > 10
            THEN
                PRINT "OUT OF RANGE"
                EXIT DEF
            ELSE
                FN.DISCONT = A^A
            END IF
        END DEF
50 INPUT Z
60 PRINT FN.DISCONT(Z)
70 END

RUNNH

? 12
OUT OF RANGE
 0
```

In this example, the defining expression is in the ELSE clause. This assigns a value to the function if A is less than 10. Because A is greater than 10, BASIC prints "OUT OF RANGE" and executes the EXIT DEF statement. The function returns zero because control is transferred to the END DEF statement before a value was assigned. Thus this example tests the arguments before the function is evaluated.

If you do not explicitly declare the function with the DECLARE statement, the restrictions for naming a multi-line DEF are the same as those for the single-line DEF. However, explicitly declaring a DEF function can make a program easier to read and understand. For example:

```
100     DECLARE STRING FUNCTION CONCAT (STRING, STRING) !Declare the function
            .
            .
            .

1000    DEF STRING CONCAT (STRING Y, STRING Z)
1100    CONCAT = Y + Z  !Define the function
1200    FNEND
            .
            .
            .
20000   NEW_STRING$ = CONCAT(A$, B$) !Invoke the function
            .
            .
            .
32767   END
```

This example merely concatenates two strings. The following example returns a number in a specified modulus:

```
100     DECLARE REAL FUNCTION MDLO (REAL, INTEGER)
200     DEF MDLO( REAL  ARGUMENT, INTEGER MODULUS )

        !Check for argument equal to zero

        EXIT DEF IF ARGUMENT = 0

        !Check for modulus equal to zero, modulus equal to absolute
        !value of argument, and modulus greater than absolute
        !value of argument.

        SELECT MODULUS
        CASE = 0%
                EXIT DEF
        CASE > ABS( ARGUMENT )
                EXIT DEF
        CASE = ABS( ARGUMENT )
                MDLO = ARGUMENT
                EXIT DEF
        END SELECT

300     !If argument is negative, set flag NEGATIVE% and set ARGUMENT
        !to its absolute value.

        IF ARGUMENT < 0
                THEN ARGUMENT = ABS( ARGUMENT )
                     NEGATIVE%  = -1%
```

```
400     UNTIL ARGUMENT < MODULUS

              ARGUMENT = ARGUMENT - MODULUS

        !If this calculation ever results in zero, MDLO returns zero

              IF ARGUMENT = MODULUS
              THEN MDLO = 0
                   EXIT DEF
              END IF
        NEXT

500     !ARGUMENT now contains the right number, but the sign may be wrong.
        !If the negative argument flag was set, make the result negative.

        IF NEGATIVE%
              THEN MDLO = - ARGUMENT
              ELSE MDLO = ARGUMENT

600     END DEF
        !.
        !.
        !.
32767   END
```

Because these functions are declared in DECLARE statements, the function names need not conform to traditional BASIC usage.

The following is an example of a recursive function:

```
100 DECLARE INTEGER FUNCTION FACTOR ( INTEGER )
200 DEF INTEGER FACTOR ( INTEGER F )
        IF F <= 0%
            THEN FACTOR = 1%
            ELSE FACTOR = FACTOR(F - 1%) * F
        END IF
    END DEF
```

This example defines a function that returns a number's factorial value.

Any variable accessed or declared in the DEF and not in the formal parameter list is global to the program unit. When BASIC evaluates the user-defined function, these global variables contain the values last assigned to them in the surrounding program module.

To prevent confusion, variables declared in the formal parameter list should not appear elsewhere in the program. Note that, if your function definition actually uses global variables, these variables cannot appear in the formal parameter list.

You cannot transfer control into a multi-line DEF except by invoking it. You should not transfer control out of a DEF except by way of an EXIT DEF or END DEF statement. This means that:

- If the DEF contains an ON ERROR GOTO, GOTO, ON GOTO, GOSUB, ON GOSUB, or RESUME statement, that statement's target line number must also be in that DEF.
- An ON ERROR GO BACK statement can transfer control out of a DEF; however, a RESUME statement in an error handler outside the DEF cannot transfer control back into the DEF.
- A subroutine cannot be shared by more than one DEF. However, you can rewrite the subroutine as a DEF with no parameters. Other function definitions can share this type of DEF.

A DEF function never changes the value of a parameter passed to it. Further, because formal parameters are local to the function definition, a reference to one of these variables outside of the DEF statement creates a new variable. For example:

```
10      DEF FN.SUM%(FIRST%,SECOND%) = FIRST% + SECOND%
20      A% = 50%
30      B% = 25%
40      PRINT FN.SUM%(A%, B%)
50      PRINT FIRST%
60      END

RUNNH

 75
 0
```

Because it is outside the function definition, line 50 creates a new variable (initialized to zero), and BASIC prints this value.

6.4 External Functions

An external function is a separately compiled program module that returns a value. To create the function subprogram, you use the FUNCTION, END FUNCTION, and EXIT FUNCTION statements. The difference between a FUNCTION subprogram and a SUB subprogram is that the FUNCTION subprogram returns a value.

External functions are useful because they:

- Can be invoked by any program module
- Allow up to 255 parameters in VAX–11 BASIC, and up to 8 parameters in BASIC–PLUS–2
- Allow arrays to be passed as parameters
- Allow more than one value to be returned by modifiable parameters

You use the EXTERNAL statement to name and explicitly declare:

- The data type returned by the external function
- The data type of the formal parameters
- The parameter passing mechanism of the formal parameters (VAX–11 BASIC only)

After you've created an external function, you compile it and link (or task-build) it with the program modules that invoke it. The syntax for invoking an external function is the same as for invoking a built-in or DEF function.

6.4.1 FUNCTION, EXIT FUNCTION, and END FUNCTION Statements

The FUNCTION statement marks the beginning of a FUNCTION subprogram. Its format is:

```
FUNCTION data-type func-nam [(formal-list)]
```

where:

data-type Is a data-type keyword specifying the data type of the return value. In VAX–11 BASIC, data-type can also be a RECORD name.

func-nam Is the name of the function. This name can be up to 6 characters for BASIC–PLUS–2 and up to 30 characters for VAX–11 BASIC.

formal-list Is an optional list of formal parameters. These parameters are variables used to calculate the value the function returns. Each formal parameter can be preceded by a data-type keyword. In VAX–11 BASIC, each formal parameter can be followed by a parameter passing mechanism.

The END FUNCTION statement: 1) marks the end of a function subprogram, 2) returns a value, and 3) returns program control to the statement that invoked the function. Its format is:

```
END FUNCTION
```

FUNCTIONEND is a synonym for END FUNCTION, but END FUNCTION is preferred.

The EXIT FUNCTION statement immediately returns program control to the statement that invoked the function. It is equivalent to an unconditional transfer to the FUNCTIONEND statement. Its format is:

```
EXIT FUNCTION
```

FUNCTIONEXIT is a synonym for EXIT FUNCTION, but EXIT FUNCTION is preferred. Here is an example of an external function:

```
100     FUNCTION REAL SPHERE_VOLUME (REAL R)
200     IF R <= 0 THEN EXIT FUNCTION
300     SPHERE_VOLUME = 4/3 * PI * R ** 3
400     END FUNCTION
```

This function returns the volume of a sphere of radius R. If this function is invoked with an actual parameter value less than or equal to zero, the function returns zero.

Because external functions are subprograms, you can pass modifiable parameters (including entire arrays) to them. See *BASIC on VAX/VMS Systems*, *BASIC on RSX–11M/M–PLUS Systems* or *BASIC on RSTS/E Systems* for more information on subprograms and modifiable parameters.

6.4.2 EXTERNAL Statement

Function subprograms must be declared with the EXTERNAL statement. When used in this way, the EXTERNAL statement's format is:

```
EXTERNAL data-type FUNCTION {func-nam[([external-param],...)]},...
```

```
pass-mech: { BY DESC
           { BY REF
           { BY VALUE }
```

```
external-param: [data-type] [DIM([,]...)] [=int-const] [pass-mech]
```

where:

data-type Is a BASIC data-type keyword. In VAX–11 BASIC, data-type can also be a RECORD name.

func-nam Is the name of the function. This name can be up to 6 characters for BASIC–PLUS–2 and up to 30 characters for VAX–11 BASIC.

DIM ([,]...) Specifies that the parameter is an array. You specify the number of dimensions in the array with commas; no commas specifies a one-dimensional array, one comma specifies a two-dimensional array, and so on.

int-const Specifies the length of a string parameter, or the length of each element in a string array parameter. Specifying string lengths for parameters is valid only in VAX–11 BASIC.

In VAX–11 BASIC, each parameter can be followed by a BY clause, specifying a parameter passing mechanism. The following example shows the declaration and invocation of the external function defined in the previous example:

```
1000     EXTERNAL REAL FUNCTION SPHERE_VOLUME (REAL)
1100     TEMP_VOLUME = SPHERE_VOLUME(5.925)
1200     PRINT TEMP_VOLUME
```

Note that the EXTERNAL statement specifies only the data type of the parameter; it does not specify a formal or actual parameter. Whenever you invoke an external function, BASIC converts the actual parameter to the data type specified in the EXTERNAL statement.

Note that this module is compiled separately from the FUNCTION subprogram. You can link or task-build these modules together to run the program from monitor level. To run the program in the BASIC environment, you: 1) compile the function subprogram, 2) load the resulting object module with the LOAD command, 3) read in the main program with the OLD command, and 4) type RUN.

See *BASIC on VAX/VMS Systems*, *BASIC on RSX–11M/M–PLUS Systems*, or *BASIC on RSTS/E Systems* for more information on linking subprograms

Caution

Ensure that the parameter data types specified in the EXTERNAL statement agree with those specified in the FUNCTION statement, or the function will produce unexpected results.

Chapter 7
Arrays

This chapter describes how to create and use arrays. Arrays can have up to 8 dimensions in PDP–11 BASIC–PLUS–2 and up to 32 dimensions in VAX–11 BASIC. Arrays can be redimensioned at run time. In addition, you can use data-type keywords to specify the type of data the array contains.

An array is a set of data ordered in any number of dimensions. A one-dimensional array is called a list or vector. A two-dimensional array is called a matrix. Subscripts define the position of an element in an array. When you create an array, you specify the number of dimensions and the maximum subscript value, or *bounds*, in each dimension. Thus, the bounds specified when the array is created determine the size of the array.

BASIC arrays are zero-based; when calculating the number of elements in a dimension, you count from zero to the number of elements specified.

You can create arrays either implicitly or explicitly. You implicitly create arrays having any number of dimensions by referencing an element of the array. You create arrays explicitly by declaring them in a DIM, DECLARE, COMMON, or MAP statement.

You can use MAT statements to create and manipulate arrays. However, you can use MAT statements only on arrays of one or two dimensions.

7.1 Creating Arrays Explicitly

You can create arrays explicitly with four BASIC statements:

- DECLARE
- DIM[ENSION] [#]
- COMMON
- MAP

Normally, you can use the DECLARE statement to create arrays. However, in certain cases, you may want to create the array with another statement:

- You use the DIM statement to create virtual arrays and arrays that can be redimensioned at run time.
- You use the COMMON statement to create arrays that can be shared among program modules or to create arrays of fixed-length strings.
- You use the MAP statement to create an array and associate it with a record buffer, or to overlay the storage for an array, thus accessing the same storage in different ways.

When you create an array, bounds determine the array's size. The maximum value allowed for a bound can be as large as 32767 (on PDP–11 systems) or 2147483467 (on VAX/VMS systems). However, it is important to note that there is a limit to the amount of storage your system can allocate. Although BASIC allows bound values within these ranges, very large arrays can cause an internal allocation error. Also, arrays of more than two dimensions require a surprising amount of storage. For example, an array with bounds of (1,1,1,1,1,1,1,1) contains 256 elements.

The following sections explain the use of arrays with these statements.

7.1.1 Creating Arrays with the DECLARE Statement

The DECLARE statement creates and names variables and arrays. When used to create an array, its format is:

```
DECLARE data-type array-nam(b1 [,b2. . .,bn]). . .
```

where:

data-type	Is a valid BASIC data-type keyword or a RECORD name (VAX–11 BASIC only).
array-nam	Is the name of the array.
b1...bn	Are bounds. Arrays defined in DECLARE statements can have only constant values as bounds. The number of bounds determines the number of dimensions the array has; the value of each bound determines the maximum subscript value in that dimension. Note that the DECLARE statement does not allow implicit data typing; you must specify a data-type keyword when creating arrays with DECLARE.

All elements of arrays created with the DECLARE statement are initialized to zero or the null string. For example:

```
100     DECLARE LONG FIRST_ARRAY(10)
```

This statement creates a longword integer array with 11 elements. Each element has an initial value of zero.

Although BASIC initializes array elements to zero or the null string, it is good programming practice to initialize all array elements by assigning them values in your program. For example, this program creates a three-dimensional string array and initializes each element to the null string.

```
100     DECLARE STRING SECOND_ARRAY(10,10,10)
200     DECLARE LONG LOOP_1, LOOP_2, LOOP_3
300     FOR LOOP_1 = 0% TO 10%
            FOR LOOP_2 = 0% TO 10%
                FOR LOOP_3 = 0% TO 10%
                    SECOND_ARRAY(LOOP_1, LOOP_2, LOOP_3) = ""
                NEXT LOOP_3
            NEXT LOOP_2
        NEXT LOOP_1
```

This program creates a 1331–element array of dynamic strings and initializes each element to the null string.

Note data type STRING causes the creation of an array of dynamic strings. To create an array of fixed-length strings, declare the array in a COMMON or MAP statement.

7.1.2 Creating Arrays with the DIM Statement

The DIM statement creates and names an array or arrays. You should use the DIM statement to create an array only when you want to: 1) redimension the array at run time or 2) create a virtual array. When creating arrays with DIM, you specify the type of data the array holds by either: 1) a data-type keyword or 2) a special suffix on the array name. The DIM statement format is:

DIM[ENSION] [#int-const,] data-type array-nam(b1 [,b2. . .,bn]). . .

where:

#int-const	Is the channel number of an open file on which you want to create a virtual array.
data-type	Is any valid BASIC data-type keyword or RECORD name (VAX–11 BASIC only).
array-nam	Is the name of the array.
b1...bn	Are bounds. Arrays in DIM statements can have either constants or variables as bounds. The number of bounds determines the number of dimensions the array has; the value of each bound determines the maximum subscript value in that dimension.

The array name can be any valid variable name. If you do not supply a data-type keyword, the data type is determined by the suffix of the array name:

- If the array name ends in a dollar sign, the array stores string data.
- If the array name ends in a percent sign, the array stores integer data.
- If the array name ends in neither a percent sign nor a dollar sign, the array stores data of the default type. The default type is single-precision floating-point unless you change the default. See Chapter 5 for more information on default data types.

If the DIM statement contains a data-type keyword, the array name cannot end in a dollar sign or percent sign.

The DIM statement can be either executable or declarative. If the specified bounds are constants, the DIM statement is declarative. This means that the storage is allocated at compile time, and the array cannot appear in any other DIM statement.

However, if any of the specified bounds are variables (simple or subscripted) the DIM statement is executable. This means that the storage for the array is allocated at run time, and the array can be redimensioned with a DIM statement any number of times.

Note

In the DIM statement, bounds can be either constants or variables (simple or subscripted), but not expressions.

When an array is redimensioned with the executable DIM statement, the array can become larger or smaller than it was. However, redimensioning an array in this way causes it to be reinitialized, and all data in the array is lost. In contrast, MAT statements let you redimension an array to be the same size or smaller than it was. However, MAT statements redimension arrays only when assigning values or performing matrix I/O; therefore, the fact that MAT reinitializes the array does not matter. See Section 7.3 for more information on MAT statements.

These restrictions apply to arrays:

- When referencing an array, you must use the same number of subscripts as was specified in the DIM statement.
- You can use identical names for a simple variable and an array; for example, A% and A%(5,5). However, this is not recommended programming practice. If you use identical names for arrays with a different number of subscripts, for example, A(5), and A(10,10), BASIC prints the warning error "Inconsistent subscript usage" at compile time.
- If you reference an array element whose subscripts are larger than the subscripts specified in the last execution of the DIM, or less than zero, BASIC signals the error "Subscript out of range" (ERR=55).

7.1.2.1 Declarative DIM Statements

Declarative DIM statements are those with positive integer constants as bounds. The percent sign is optional for bounds; however, BASIC signals "Integer constant required" if a constant bound contains a decimal point. This is an example of a declarative DIM:

```
100     DIM #1%, STRING VIRT_ARRAY(100) = 256%
```

This statement creates a 101–element virtual array containing string data. The elements of this array can have a maximum length of 256 characters. These restrictions apply to the use of declarative DIM statements:

- A declarative DIM statement must lexically precede any reference to the array it dimensions.
- For declarative DIM statements, if you reference an array element whose subscripts are larger than the subscripts specified in the DIM statement, BASIC signals the error "Subscript out of range" (ERR=55).
- Because a declarative DIM allocates storage at compile time, an array of this type cannot appear in any other declarative statement such as COMMON, MAP, DECLARE, or a later DIM statement.

7.1.2.2 Executable DIM Statements

Executable DIM statements are those with at least one variable bound. Bounds can be constants or simple variables, but at least one bound must be a variable. Executable DIM statements let you

redimension an array at run time. The array can become larger or smaller, but the number of bounds cannot change. That is, you cannot redimension a four-dimensional array to be five-dimensional.

The executable DIM statement cannot be used on arrays in COMMON, MAP, DECLARE or declarative DIM statements, nor on virtual arrays or arrays received as formal parameters.

An executable DIM statement always reinitializes the array to zero (for numeric arrays) or the null string (for string arrays), thus destroying any data that was in the array. For example:

```
100       X% = 10
200       DIM REAL_ARRAY(X%)
300       REAL_ARRAY(1%) = 100
400       PRINT REAL_ARRAY(1%)
500       DIM REAL_ARRAY(X%)
600       PRINT REAL_ARRAY(1%)
700       END
RUNNH

 100
 0
```

The DIM statement in line 500 reinitializes REAL_ARRAY; thus, REAL_ARRAY(1%) equals zero in line 600.

You cannot reference any array element until the array has been dimensioned with the executable DIM statement. An attempt to do so causes BASIC to signal "Subscript out of range" (ERR = 55).

7.1.3 Creating Arrays with the COMMON Statement

You should create arrays with COMMON when you need an array of fixed-length strings, or when you want to share an array among program modules. When used to create arrays, the COMMON statement format is:

COM[MON] [(com-nam)] [data-type] array-nam(b1 [,b2. . .,bn]) [= int-const]

where:

com-nam	Is the name of the COMMON block.
data-type	Is any valid BASIC data-type keyword or RECORD name (VAX–11 BASIC only).
array-nam	Is the name of the array.
b1...bn	Are bounds. Arrays in COMMON statements can have only constants as bounds. The number of bounds determines the number of dimensions the array has; the value of each bound determines the maximum subscript value in that dimension.
int-const	Specifies the length of each element in a string array.

Program modules can share arrays in COMMON statements by defining a COMMON block with the same name. For example:

Main Program

```
100       COMMON (ABC) STRING ACCESS_LIST(100) = 10%
```

Subprogram

```
100       SUB SUB1
200       COMMON (ABC) STRING NEW_LIST(100) = 10%
```

The COMMON statements in these programs create a 101–element array of fixed-length strings, each element 10 characters long. Because the main and subprograms use the same COMMON name, the storage for these arrays is overlaid when the programs are linked or task-built. Thus, both programs can read and write data to the array.

7.1.4 Creating Arrays with the MAP Statement

You should create arrays with MAP only when you want the array to be part of a record buffer, or when you want to overlay the storage containing the array. Note that string arrays in MAPs are always fixed-length. When used to create arrays, the MAP statement format is:

```
map (map-nam) [data-type] array-nam(b1[,b2...,bn]) [=int-const]
```

where:

map-nam	Is the name of the MAP. MAP statements require a map-nam.
data-type	Is any valid BASIC data-type keyword or RECORD name (VAX–11 BASIC only).
array-nam	Is the name of the array.
b1...bn	Are bounds. Arrays in MAP statements can have only constants as bounds. The number of bounds determines the number of dimensions the array has; the value of each bound determines the maximum subscript value in that dimension.
int-const	Specifies the length of each element in a string array.

You associate the array with a record buffer by naming the MAP in the MAP clause of the OPEN statement. For example:

```
100     MAP (ABC) STRING TEAM(10) = 20, WORD BOWLING_SCORES(32)
200     OPEN "BOWLING.DAT" AS FILE #1%, SEQUENTIAL VARIABLE, &
                        MAP ABC
```

The MAP statement creates two arrays: an 11–element fixed-length string array named TEAM and a 33–element array of WORD integers named BOWLING_SCORES. Because the OPEN statement specifies MAP ABC, the storage for these arrays is used as the record buffer for the open file.

7.2 Creating Arrays Implicitly

There are two ways to create implicit arrays: 1) by referencing an element of an array that has not been explicitly declared and 2) with MAT statements. When BASIC first creates an implicit array, the maximum subscript value is 10, although you can redimension implicit arrays with an executable DIM statement.

An array created by referencing an element can have up to 8 dimensions in PDP–11 BASIC–PLUS–2 and up to 32 dimensions in VAX–11 BASIC. An array created with a MAT statement can have only one or two dimensions.

Note

The ability to create arrays implicitly exists for compatibility with previous implementations of BASIC. However, it is much better programming practice to explicitly declare all arrays before using them.

If you reference an element of an array that has not been explicitly declared, BASIC creates a new array with the name you specify. Arrays created by reference have default subscripts of (10), (10,10), (10,10,10) and so on, depending on the number of dimensions specified in the array reference. For example:

```
10 LET A(5,5,5) = 3.14159
20 LET B%(3) = 33
30 LET C$(2,2) = "Shirley Merkel"
40 END
```

This program implicitly creates three arrays by reference and assigns a value to one element of each.

- Line 10 creates an 11–by–11–by–11 array that stores floating-point numbers and assigns the value 3.14159 to element (5,5,5).
- Line 20 creates an 11–element list that stores integers and assigns the value 33 to element (3).
- Line 30 creates an 11–by–11 string array and assigns the value "Shirley Merkel" to element (2,2).

When you create an implicit numeric array by referencing an element, BASIC initializes all elements (except the one assigned a value) to zero. For implicit string arrays, BASIC initializes all elements (except the one assigned a value) to a null string. When you create an array by reference, you cannot specify a subscript greater than 10. An attempt to do so causes BASIC to signal "subscript out of range" (ERR = 55).

Note that you cannot create an array implicitly, then redimension the array with a executable DIM statement. The DIM statement must execute before any reference to the array.

An implicit array cannot appear in a declarative DIM statement. For example, the first program is valid while the second is not:

Example 1

```
100     DIM NEW_ARRAY(15,10,5)
200     NEW_ARRAY(5,5,5) = 1
```

Example 2

```
100     NEW_ARRAY(5,5,5) = 1
200     DIM NEW_ARRAY(15,10,5)
```

In the first program, line 100 dimensions an array with a declarative DIM statement. Line 200 assigns a value to an element of that array. In the second program, line 100 creates an implicit array. The program fails because the DIM statement on line 200 is declarative (it contains only constant subscripts).

7.3 MAT Statements

MAT statements let you assign values to or display entire arrays with a single statement. They also:

- Implicitly create arrays
- Assign names to arrays
- Specify array dimensions

- Redimension existing arrays (to equal or smaller sizes)
- Assign element values
- Print the contents of arrays
- Perform matrix arithmetic

Note

MAT statements cannot be used on arrays of more than two dimensions, on DECIMAL arrays, nor on arrays appearing in RECORD instances (the DECIMAL data-type and RECORD statements are available only in VAX–11 BASIC). Further, when MAT statements execute, they use row and column zero to store intermediate calculations. This means that MAT statements can overwrite data stored in row and column zero, and you should not depend on data in these elements if your program uses MAT statements.

The default subscripts for arrays created implicitly with MAT statements are (10) or (10,10). The default is two dimensions. This means that, if you create an array with a MAT statement and do not specify any subscripts, BASIC creates a two-dimensional, 11–by–11 array. If you specify a single subscript, BASIC creates a one-dimensional array with 11 elements.

Table 7–1 lists MAT statements and explains their function.

Table 7–1: MAT Statements

Statements	Function
MAT	Assigns values of zero, one, or a null string to array elements. Also copies the values of one array to another and performs matrix arithmetic.
MAT INPUT [#]	Assigns values to array elements from your terminal or a terminal-format file.
MAT LINPUT [#]	Assigns string values to string array elements from your terminal or from a terminal-format file.
MAT PRINT [#]	Displays the contents of an array on your terminal, or writes array element values to a terminal-format file.
MAT READ	Assigns DATA statement values to array elements.

For example:

```
10 MAT Z.ARRAY$ = NUL$(7,7)
20 MAT Z.ARRAY$ = NUL$(5,5)
30 MAT A = B + C
40 END
```

Line 10 creates the string array Z.ARRAY$ with eight rows and eight columns and assigns a null string to all elements. Line 20 redimensions the array to six rows and six columns. Line 30 adds the values in each corresponding element of arrays B and C and stores the values in the corresponding elements of array A.

7.3.1 Assigning Values to Array Elements

You can assign values to array elements: 1) from within your program, 2) from an external source, such as the terminal or files, or 3) with MAT statements.

You can assign values within your program with:

- The LET statement
- The MAT statement
- The MAT READ statement

7.3.1.1 Assigning Values with the LET Statement

The LET statement assigns values to individual array elements:

```
100 DIM VOUCHER.NUM%(100)
 . . . . . .
3520 LET VOUCHER.NUM%(20) = 3253%
 . . . . . .
32767 END
```

You can also assign values to a portion of an array with the LET statement and a FOR–NEXT loop:

```
100 DIM PO.NUMBER%(100,100)
 . . . . . .
2040 FOR I% = 1% TO 3%
2050     FOR J% = 5% TO 10%
2060
             LET PO.NUMBER%(I%,J%) = 0%
2070     NEXT J%
2080 NEXT I%
 . . . . . .
32767 END
```

The FOR/NEXT loop in lines 2040 through 2080 assigns zero to array elements (1,5) through (1,10), to (2,5) through (2,10), and to (3,5) through (3,10).

7.3.1.2 Assigning Values with the MAT Statement

The MAT statement can create an array and optionally assign values to all elements. Its format is:

```
MAT array-name = mat-keyword [(subscript[,subscript])]
```

The array name can specify an existing array. MAT statements do not assign values to row and column zero.

Table 7–2 lists MAT statement keywords and their functions.

Table 7–2: MAT Statement Keywords

MAT Keyword	Function
ZER	Sets the value of all elements in a numeric array to zero.
CON	Sets the value of all elements in a numeric array to one.
IDN	Sets the array to the identity matrix, that is, it sets the value of all elements in real or integer arrays to zero, except for those elements on the diagonal from element (1,1) to element (n,n), where n is the largest subscript in the array. The elements on the diagonal are set to one. IDN applies to square arrays only.
NUL$	Sets the value of all elements in a string array to a null string.

The MAT statement does not require subscripts. For existing arrays:

- If you do not specify subscripts, BASIC does not change the current subscripts.
- If you specify subscripts, BASIC redimensions the array to the specified subscripts. When redimensioning arrays with MAT, you cannot increase the total number of array elements (including those in row and column zero).

When creating arrays with MAT:

- If you do not supply subscripts, BASIC assigns two subscripts, each with a value of 10.
- If you specify subscripts, they define the dimensions of the array being implicitly created. Subscript values cannot exceed 10.

For example:

```
10        DIM A(10,10), B(15), C(20,20)
20        MAT A = ZER      !Sets all elements of A to 0
30        MAT B = CON(10) !Sets elements of B to 1; redimensions B
40        MAT C = IDN(10,10) !Redimensions C to 10x10 identity matrix
50        MAT PRINT A;
          PRINT
          PRINT
60        MAT PRINT B;
          PRINT
          PRINT
70        MAT PRINT C;

RUNNH

 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0          (ARRAY A)
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0

 1   1   1   1   1   1   1   1   1   1          (ARRAY B)
```

```
1  0  0  0  0  0  0  0  0  0
0  1  0  0  0  0  0  0  0  0
0  0  1  0  0  0  0  0  0  0
0  0  0  1  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0          (ARRAY C)
0  0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  1  0  0
0  0  0  0  0  0  0  0  1  0
0  0  0  0  0  0  0  0  0  1
```

7.3.1.3 Assigning Values with the MAT READ Statement

The MAT READ statement assigns values from DATA statements to array elements. Its format is:

MAT READ array-name [(subscript [,subscript])]. . .

Subscripts are optional. They define: 1) the dimensions of the array being created or 2) the new dimensions of an existing array.

If you do not provide enough data in DATA statements to fill the specified array, BASIC leaves the remaining array elements unchanged. If you provide more data values than there are array elements, BASIC assigns enough values to fill the array and leaves the DATA pointer at the next value. For example:

```
10        MAT READ B(2,2)
20        MAT READ C(2,2)
30        PRINT
          PRINT "MATRIX B"
          PRINT
          PRINT
          MAT PRINT B;
40        PRINT
          PRINT "MATRIX C"
          PRINT
          PRINT
          MAT PRINT C;
50        DATA 1,2,3,4,5,6,7,8,9,10
60        END

RUNNH

MATRIX B

 1   2
 3   4

MATRIX C

 5   6
 7   8
```

BASIC fills matrix B with the first four DATA items, fills matrix C with the next four DATA values, and leaves the DATA pointer at the ninth value in the DATA list.

7.3.2 Assigning Values from the Terminal

You can assign values to array elements while your program is executing with:

- MAT INPUT
- MAT LINPUT

7.3.2.1 MAT INPUT Statement

The MAT INPUT statement assigns values from your terminal to array elements. Its format is:

```
MAT INPUT array-name [(subscript [,subscript])]. . .
```

The optional subscripts define: 1) the dimensions of the array being created implicitly or 2) the new dimensions of an existing array. If you are creating the array, the value of a subscript cannot exceed 10.

The MAT INPUT statement requests data from your terminal, as does the INPUT statement; it prints a question mark (?) prompt. However, you cannot include a string prompt with the MAT INPUT statement.

In response to the question mark, enter a series of values separated by commas. BASIC enters the values you type into successive array elements by row, starting with element (1,1) and filling row 1 before starting row 2. If you provide fewer data items than there are elements, the remaining elements are unchanged.

For example:

```
10        MAT INPUT A
20        PRINT
          MAT PRINT A;
30        MAT INPUT B(2,2)
40        PRINT
          PRINT
          MAT PRINT B;

RUNNH

? 1,2,3,4,5,6,7,8,9,10,11,12,13

  1   2   3   4   5   6   7   8   9   10
 11  12  13   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0


? 1,2,3,4


 1   2
 3   4
```

BASIC creates a 10–by–10 matrix for Array A by default. The values supplied by the MAT INPUT statement fill the first 13 elements, filling row 1 before beginning row 2. The remainder of the matrix elements contain zeros. Lines 30 and 40 redimension array B to 9 elements, provide values for its elements, and display the results. If you provide more items than there are elements, BASIC ignores the excess.

The MAT INPUT statement can also redimension an existing array. For example:

```
10 DIM NEW.ARRAY%(5,5)
20 MAT INPUT NEW.ARRAY%(2,4)
30 MAT PRINT NEW.ARRAY%;
40 END

RUNNH

? 1,2,3,4,5,6,7,8

 1  2  3  4
 5  6  7  8
```

When entering values in response to MAT INPUT, you can enter an ampersand as the last character on the line and continue on the next line.

7.3.2.2 MAT LINPUT Statement

The MAT LINPUT statement assigns string values to string array elements. Its format is:

MAT LINPUT {array-name [(subscript [, subscript])]} ,...

The optional subscripts define: 1) the dimensions of the array being created or 2) the new dimensions of an existing array. If you are creating the array, subscript values cannot exceed 10.

The MAT LINPUT statement prompts for individual array elements. It fills the array by rows, starting with element (1,1). It assigns the line you type (including commas, semicolons, and quotation marks) to an array element; however, the line terminator is not included. For example:

```
10      DIM EMP.NAM$(5,5)
20      MAT LINPUT EMP.NAM$(2,2)
30      PRINT EMP.NAM$(1,1)
40      PRINT EMP.NAM$(1,2)
50      PRINT EMP.NAM$(2,1)
60      PRINT EMP.NAM$(2,2)
70      END

RUNNH

? HODGES
? LAFFERTY
? ELDON
? HOPKINS
HODGES
LAFFERTY
ELDON
HOPKINS
```

By specifying the subscripts (2,2) in line 20, MAT LINPUT redimensions the array to four elements and then prompts for these elements.

7.3.3 Assigning Values from Terminal-Format Files

Two statements transfer data from a terminal-format file to an array:

- MAT INPUT #
- MAT LINPUT #

7.3.3.1 MAT INPUT # Statement

The MAT INPUT # statement reads data from a terminal-format file and writes it to an array. Its format is:

```
MAT INPUT #chnl-exp, array-name [(int-exp [,int-exp])]
```

The channel expression specifies an open terminal-format file. The array name specifies the array to which the values are assigned.

The MAT INPUT # statement takes values from the file and assigns them to the matrix elements by rows, starting with element (1) or (1,1). It fills the elements in row 1 before starting row 2. The file can have one or more values in each record, however, if multiple values must be separated with commas.

If there is more data in the file than there are array elements, BASIC ignores the excess. If there is less, the remaining elements are unchanged.

7.3.3.2 MAT LINPUT # Statement

The MAT LINPUT statement reads string values from a terminal-format file and writes them to a string array. Its format is:

```
MAT LINPUT #chnl-exp, {str-arr-nam [(subs [,subs])]},...
```

where:

#chnl-exp	Is the channel number of an open terminal-format file.
str-arr-nam	Is the name of the string array to receive the data.
subs	Is a subscript. The subscripts determine how much data is assigned to the array.

MAT LINPUT excludes line terminators when assigning values to string array elements.

If there are more values in the file than there are array elements, BASIC ignores the excess. If there are fewer, BASIC assigns a null string to the remaining elements. For example:

```
10 OPEN "PARTS.DAT" FOR INPUT AS FILE #1%
20 DIM PART_NAME$(50)
30 MAT LINPUT #1%, PART_NAME$
. . . . .
```

This program reads 50 records from the disk file PARTS.DAT to the array named PART_NAME$.

7.4 Array Output

You can write data from an array with:

- The PRINT statement
- The MAT PRINT statement
- The MAT PRINT # statement

7.4.1 PRINT Statement

You print individual array elements by naming those elements in the PRINT statement. For example:

```
 . . . . . .
310 PRINT PARTS.LIST$(35)
 . . . . . .
```

With a FOR–NEXT loop, you can print all or part of an array:

```
100 DIM CAPTURE.RATIO(10,10)
 . . . . . .
10030 FOR Y% = 7% TO 10%
10040   FOR X% = 7% TO 10%
10050       PRINT CAPTURE.RATIO(X%,Y%)
10060   NEXT X%
10070 NEXT Y%
 . . . . . .
32767 END
```

7.4.2 MAT PRINT Statement

The MAT PRINT statement prints some or all of an array's elements, excluding row and column zero. Its format is:

MAT PRINT array-name [(subs [,subs])]

where:

array-name	Is the name of the array.
subs	Is a subscript.

Subscripts are optional. If you do not specify subscripts, MAT PRINT displays the entire array, excluding row and column zero. If you specify subscripts, MAT PRINT displays that subset of the array. The MAT PRINT statement does not redimension an existing array.

If the last character in the MAT PRINT array list is a:

Semicolon	BASIC begins each array on a separate line. Data values on each line are packed together.
Comma	BASIC begins each array row on a separate line and each data value in a separate print zone.

If there is neither a comma nor semicolon after the array name, BASIC prints each array element on a separate line. For example:

```
10 MAT INPUT A(5)
15 PRINT
20 MAT PRINT A
25 PRINT
30 MAT PRINT A, A;
60 END
```

(continued on next page)

```
RUNNH
? 5

  0
  0
  0
  0
  5

  0           0           0           0           5

  0  0  0  0  5
```

7.4.3 MAT PRINT # Statement

The MAT PRINT # statement reads values from an array and writes them to a terminal-format file. Its format is:

```
MAT PRINT #chnl-exp, {array-name [(subs [,subs])]},...
```

where:

#chnl-exp	Is the channel number of an open terminal-format file.
array-name	Is the name of the array that holds the data.
subs	Is a subscript. The subscripts determine how many array elements are written to the file.

MAT PRINT # takes values by row, starting with element (1) or (1,1), and writes each element to a sequential record in the terminal-format file.

7.4.4 Matrix I/O Functions (NUM and NUM2)

MAT statements do not signal error messages when:

- There are more data items than array elements to contain them.
- There are fewer data items than array elements to contain them.

BASIC provides two functions that let you determine how much data these statements transfer.

For two-dimensional arrays, the NUM function returns an integer value specifying the row number of the last data item transferred. For one-dimensional arrays, the NUM function returns the number of items entered. Its format is:

```
NUM
```

The NUM2 function returns: 1) for two-dimensional arrays, an integer value specifying the column number of the last data item transferred, and 2) for one-dimensional arrays, a zero. Its format is:

```
NUM2
```

With these functions, you can determine the number of items transferred from a terminal-format file. For example:

```
10 OPEN "EMP.DAT" FOR INPUT AS FILE #3%
20 DIM EMP.NAME$(5,5)
30 MAT INPUT #3%, EMP.NAME$
40 PRINT NUM, NUM2
50 END
```

7.5 Matrix Operators

BASIC provides a special set of MAT statements for array computations. These statements enable you to add, subtract, and multiply matrices, and to assign values to elements. If you specify an array without subscripts (for example, MAT A), the default is two dimensions.

BASIC also provides matrix functions to transpose and invert matrices, and to find the determinant of a matrix you invert.

Note

MAT operators do not operate on elements in row or column zero.

7.5.1 Arithmetic Matrix Operations

MAT operators perform matrix:

- Assignment
- Addition
- Subtraction
- Multiplication

All of these operations use the keyword MAT followed by an expression. If the array has not been previously dimensioned, these operations create an array. (The created output array's dimensions depend on the operation performed, but must be (10,10) or smaller.)

Note

You can use the MAT operators on arrays larger than (10,10) if the input and output arrays are explicitly created or received as a formal parameter.

7.5.1.1 Assignment

You can assign all values in one array to another array with the keyword MAT. For example:

```
10 MAT NEW.ARRAY = OLD.ARRAY
```

This example sets each element of NEW.ARRAY to the corresponding element in OLD.ARRAY. It also redimensions NEW.ARRAY to the dimensions of OLD.ARRAY.

7.5.1.2 Addition and Subtraction

To add the elements of two arrays, use the format:

MAT output-array = input-array1 + input-array2

To subtract the elements of two arrays, use the format:

MAT output-array = input-array1 – input-array2

For example:

```
10 MAT SUM.LIST% = FIRST.LIST% + SECOND.LIST%
```

The two input lists, FIRST.LIST% and SECOND.LIST%, must have identical dimensions. The elements of the new list, SUM.LIST%, equal the sum of the corresponding elements in the input lists.

This program subtracts one array from another:

```
10     DIM FIRST.ARRAY(30,30)
20     DIM SECOND.ARRAY(30,30)
30     DIM DIFFERENCE.ARRAY(30,30)
 . . . . . .
100    MAT DIFFERENCE.ARRAY = FIRST.ARRAY - SECOND.ARRAY
 . . . . . .
32767 END
```

Each element of DIFFERENCE.ARRAY is the arithmetic difference of the corresponding elements of the input arrays.

7.5.1.3 Multiplication

The format for matrix multiplication is:

MAT output-array = input-array1 * input-array2

The number of columns in input-array1 must equal the number of rows in input-array2. The output array contains the dot product of the two input arrays. For example:

```
100      DIM A(2,2), B(2,2), C(2,2)
200      A(1,1) = 1
         A(1,2) = 2
         A(2,1) = 3
         A(2,2) = 4
300      B(1,1) = 5
         B(1,2) = 6
         B(2,1) = 7
         B(2,2) = 8
400      MAT C = A * B
500      MAT PRINT C
```

You can also multiply a matrix by a scalar quantity. The format is:

MAT output-array = (num-exp) * input-array

BASIC multiplies each element of the input array by the scalar quantity you supply. The output array has the same dimensions as the input array. Enclose the scalar quantity in parentheses. For example:

```
10 DIM INCH.ARRAY(5), CM.ARRAY(5)
20 MAT READ INCH.ARRAY
30 DATA 1,12,36,100,39.37
40 MAT CM.ARRAY = (2.54) * INCH.ARRAY
50 MAT PRINT CM.ARRAY,
60 END

RUNNH

 2.54        30.48        91.44         254         99.9998
```

Line 40 multiplies the elements of INCH.ARRAY by the inch-to-centimeter conversion factor and places these values in CM.ARRAY.

7.5.2 Matrix Functions

BASIC provides three matrix functions:

- TRN
- INV
- DET

With these functions, you can: 1) transpose matrices, 2) invert matrices, or 3) find the determinant of an inverted matrix.

7.5.2.1 TRN Function

The TRN function transposes a matrix. Its format is:

```
MAT output-array = TRN(input-array)
```

When you transpose a matrix, BASIC interchanges the array's dimensions. For example, a matrix with N rows and M columns is transposed to a matrix with M rows and N columns. The elements in the first row of the input matrix become the elements in the first column of the output matrix. You cannot transpose a matrix to itself; MAT A = TRN(A) is invalid.

This example creates a three-by-five matrix, transposes it, and prints the results:

```
10 DIM B(3,5)
20 MAT READ B
30 MAT A = TRN(B)
40 DATA 1,2,3,4,5
50 DATA 6,7,8,9,10
60 DATA 11,12,13,14,15
70 MAT PRINT B;
80 MAT PRINT A;
90 END
```

(continued on next page)

```
RUNNH

 1  2  3  4  5
 6  7  8  9  10
 11  12  13  14  15

 1   6    11
 2   7    12
 3   8    13
 4   9    14
 5   10   15
```

7.5.2.2 INV Function

The INV function inverts a matrix. Its format is:

MAT output-matrix = INV(input-matrix)

BASIC can invert a matrix only when: 1) its subscripts are identical, and 2) it can be reduced to the identity matrix by elementary row operations.

7.5.2.3 DET Function

The DET function returns the determinant of a matrix. Its format is:

DET

The DET function returns a floating-point number that is the determinant of the last matrix inverted. If you use the DET function before inverting a matrix, the value of DET is zero.

Chapter 8
Formatting Output with the PRINT USING Statement

The PRINT USING statement controls the appearance and location of data on a line of output. With it, you can create formatted lists, tables, reports, and forms.

The ability to format data is useful because the way BASIC internally represents data may not be the best way to display that data in a report or table. For example, a program may use floating-point numbers to represent dollars and cents. The PRINT statement displays floating-point numbers with up to six digits of accuracy and can place the decimal point anywhere in that 6–digit field. In contrast, PRINT USING lets you display floating-point numbers:

- Rounded to two decimal places
- Vertically aligned on the decimal point
- Preceded by a dollar sign
- With commas every third digit to the left of the decimal point

Formatting monetary values in this way makes a much more readable report. Another use for formatted numeric values might be to print checks on the computer's line printer. To do this, PRINT USING lets you print numbers with a dollar sign and an asterisk-filled field preceding the first digit.

PRINT USING also formats string data. With it you can left- and right-justify string expressions or center a string expression over a specified column position. Further, the PRINT USING statement can contain string literals. These are strings that do not control the format of a print item but instead are printed exactly as they appear in the *format string*.

The PRINT USING statement requires a format string. This format string determines the way in which the items to be printed are formatted. You specify the format string in the PRINT USING statement; it can be a literal, a variable, a named string constant, or a combination of these.

This chapter explains how to create a format string and then how to use a format string to display numeric and string data.

8.1 PRINT USING

The format of the PRINT USING statement is:

PRINT USING {str-exp}, exp [, exp. . .]

where:

str-exp Contains one or more PRINT USING format fields. Str-exp can be a string variable or a string literal. If it is a string literal, you must enclose it in double quotation marks.

exp [,exp] Is a list of items to be printed.

The specification of the format string immediately follows the USING keyword and can be:

- A string literal
- A string variable
- A named string constant
- Any combination of these

You must separate print items with commas or semicolons. Separators between print items do not affect output format as they do with the PRINT statement. However, if a comma or semicolon follows the last print item, BASIC does not return the cursor or print head to the beginning of the next line after it prints the last item in the list.

8.1.1 Creating a Format String

The PRINT USING statement requires a format string to control the appearance of the output line. The format string is made up of format fields; each format field controls the output of one print item.

You can think of the format string as an image of the PRINT USING output line. Each format field in the format string controls the appearance of one print item. Thus, BASIC scans the format string and uses the first format field to control the appearance of the first print item, the second format field to control the appearance of the second item, and so on. If all the format fields in the format string have been used and there are still items to be printed, BASIC moves the cursor or print head to the beginning of the next line and reuses the format string starting at the first format field.

You need not delimit format fields within a format string. Each format field can contain only certain characters (as described in the following sections). BASIC scans the format string character by character; as soon as it encounters a character that is invalid for the current format field, BASIC ends the current format field. The character that terminates the previous field can begin either: 1) a new format field or 2) a string literal. For example:

```
100     PRINT USING "###ABC###", 123, 345

RUNNH

123ABC345
```

The first three characters in the format string (###) make up a valid numeric format field. The fourth character (A) is invalid in a numeric format field; therefore, BASIC ends the first format field after the third character. BASIC continues to scan the format string, searching for a character that begins a format field. The first such character is the pound sign at character position seven. Therefore, the

characters at positions four, five, and six are stored as a string literal. The characters at positions seven, eight, and nine make up a second valid numeric format field.

When the statement executes, BASIC prints the first number in the list using the first format field, then prints the string literal ABC, and finally prints the second number in the list using the second format field.

Because any character not part of a format field is printed just as it appears in the format field, you can use a space or multiple spaces to separate format fields in the format string. For example:

```
100      DECLARE STRING CONSTANT FORMAT_STRING = "###.##    ###.##"
         DECLARE SINGLE A,B
200      A = 2.565
         B = 100.350
300      PRINT USING FORMAT_STRING; A; B; A; B

RUNNH

  2.57    100.35
  2.57    100.35
```

When line 300 executes, BASIC prints the value of A (rounded according to PRINT USING rules), three spaces, then the value of B. BASIC prints the three spaces because they are treated as a string literal in the format string. Notice that when BASIC reuses a format string, it begins on a new line.

Any of these format string specifications can contain multiple format fields. BASIC reuses the format string as many times as necessary to print all the items in the list. Whenever BASIC reuses a format string, it starts a new line of output.

8.1.2 Reusing the Format String

The process of reusing the PRINT USING format string is called format reversion. If BASIC reaches the end of the format string and items remain to be printed, BASIC starts again at the beginning of the format string and continues printing on the next line. The following example uses and reuses both numeric and string format fields:

```
10 PRINT USING "ITEM: 'LLLLLLLLLL PRICE: $$###,###.##"; &
          "CANOE"; 425.99; "BACKPACK"; 52.80; "CAMERA"; 325.89

RUNNH

ITEM: CANOE       PRICE:    $425.99
ITEM: BACKPACK    PRICE:     $52.80
ITEM: CAMERA      PRICE:    $325.89
```

This example shows how PRINT USING displays multiple numbers using a single format field:

```
100      PRINT USING "#####.###"; 2.0; -10.6; 110.43; 9862.36; 126.32
200      END

RUNNH

    2.000
  -10.600
  110.430
 9862.360
  126.320
```

Because the format string is reused for each print item, each print item appears on a new line.

8.2 Printing Numbers

With the PRINT USING statement, you can specify:

- The number of digits to print, thus rounding the number to a given place
- The decimal point location, thus vertically aligning numbers at the decimal point
- Special symbols, including trailing minus signs, asterisk-filled number fields, floating currency symbols, embedded commas, and E notation
- Debits and credits
- Leading zeros or leading spaces
- Blank-if-equal-to-zero fields
- That a special character is to be printed as a literal

Unlike the PRINT statement, PRINT USING does not automatically print a space before and after a number. Unless you reserve enough digit positions to contain the integer portion of the number (and a minus sign, if necessary), BASIC prints a percent sign (%) and displays the number in PRINT format.

8.2.1 Specifying the Number of Digits

You reserve places for digits by including a pound sign (#) for each digit position. If you print negative numbers, you must also reserve a place for the minus sign. For example:

```
10 PRINT USING "###",123    !Three places reserved
20 PRINT USING "#####",12345 !Five places reserved
30 PRINT USING "####",-678    !Four places reserved
40 END

RUNNH

123
12345
-678
```

If there are not enough digits to fill the field, BASIC prints spaces before the first digit:

```
10       FORMAT_STRING$ = "#####"
20       PRINT USING FORMAT_STRING$, 1
30       PRINT USING FORMAT_STRING$, 10
40       PRINT USING FORMAT_STRING$, -1709
50       PRINT USING FORMAT_STRING$, 12345
60       END

RUNNH

     1
    10
 -1709
 12345
```

If you have not reserved enough digits to print the fractional part of a number, BASIC rounds the number to fit the field:

```
10 PRINT USING "###";126.7
20 PRINT USING "#";5.9
30 PRINT USING "#";5.4
40 END

RUNNH

127
6
5
```

If you have not reserved enough places to print a number's integer portion, BASIC prints a warning message followed by the number in PRINT statement format. After BASIC prints the number, it completes the rest of the list in PRINT USING format. For example:

```
10 PRINT USING "###", 256
20 PRINT USING "##", 256
40 END

RUNNH

256
% 256
```

PRINT USING displays the number in line 10. In line 20, there are not enough places to the left of the decimal point to display a 3–digit number; therefore, BASIC prints the number in PRINT statement format, with a space before and after, but includes a warning sign (%).

8.2.2 Specifying Decimal Point Location

The decimal point's position in the format string determines the number of reserved places on either side of it. If the print item's fractional part does not use all of the reserved places to the right of the decimal point, BASIC fills with zeros. For example:

```
10      DECLARE STRING CONSTANT FM = "##.###"
20      PRINT USING FM, 15.72
30      PRINT USING FM, 39.3758
40      PRINT USING FM, 26

RUNNH

15.720
39.376
26.000
```

The following example shows how PRINT USING rounds numbers when specifying decimal point location:

```
10      PRINT USING "##.##", 25.789
20      PRINT USING "##.###", 100.2
30      PRINT USING "#.##",.999
40      END

RUNNH

25.79
% 100.2
1.00
```

If there are more fractional digits than reserved places to the right of the decimal point, BASIC rounds the number to fit the reserved places. Note that there must be enough places reserved to the left of the decimal point for the integer portion of the number. Otherwise, BASIC prints the number in PRINT format preceded by a warning sign.

BASIC always fills all reserved spaces to the left of the decimal point with specified digits, spaces, or the minus sign:

```
10       PRINT USING "##.##", 5.25
20       PRINT USING "##.##", -5.25
30       PRINT USING "###.##",-5.25
40       END

RUNNH

 5.25
-5.25
 -5.25
```

8.2.3 Printing Numbers with Special Symbols

Special symbols let you print numbers with trailing minus signs, asterisk-fill fields, floating currency symbols, commas, or E notation. You can also specify debits, credits, leading zeros, leading blanks, and blank-if-zero fields. Table 8–1 summarizes these special characters.

Table 8–1: Format Characters for Numeric Fields

Character		Effect on Format
#	pound sign	Reserves place for one digit.
.	decimal point (period)	Determines decimal point location.
,	comma	Prints a comma between every third digit to the left of the decimal point and reserves a place for one digit or comma.
**	two asterisks	Print leading asterisks before the first digit and reserve places for two digits.
$$	two dollar signs	Print a currency symbol before the first digit. They also reserve places for the currency symbol and one digit. By default, the currency symbol is a dollar sign. To change the currency symbol, refer to Section 8.2.3.3.
^^^^	four carets	Print a number in E (exponential) format and reserve four places for E notation.
–	minus sign	Prints a trailing minus sign for negative numbers. Printing a negative number in an asterisk fill or a currency field requires that the field also have a trailing minus sign or credit/debit character.
<0>	Zero in angle brackets	Prints leading zeros instead of leading spaces.
<%>	Percent sign in angle brackets	Prints all spaces in the field if the value of the print item, when rounded to fit the numeric field, is zero.
<CD>	CD in angle brackets	Prints credit and debit characters immediately following the number. BASIC prints CR for negative numbers and zero, and DR for positive numbers.
_	Underscore	Specifies that the next character is a literal, not a formatting character.

8.2.3.1 Commas

You can place a comma anywhere in a number field to the left of the decimal point and to the right of the field's first character. A comma cannot start a format field. BASIC then prints a comma to the left of every third digit from the decimal point. If there are fewer than four digits to the left of the decimal point, BASIC omits the comma. For example:

```
10      PRINT USING "##,###",10000
20      PRINT USING "##,###",759
30      PRINT USING "$$#,###.##",25694.3
40      PRINT USING "***,###",7259
50      PRINT USING "####,#,##",25239
60      END

RUNNH

 10,000
    759
$25,694.30
**7,259
 25,239.00
```

8.2.3.2 Asterisk Fill Fields

To print asterisks (*) before the first digit of a number, start the field with two asterisks. For example:

```
5       DECLARE STRING CONSTANT FM = "****.##"
10      PRINT USING FM, 1.2
20      PRINT USING FM, 27.95
30      PRINT USING FM, 107
40      PRINT USING FM, 1007.5
50      END

RUNNH

***1.20
**27.95
*107.00
1007.50
```

Note that asterisks reserve two places as well as cause asterisk fill.

Specify a negative number in an asterisk-fill field by using a trailing minus sign in the field. The trailing minus sign must be the last character in the format string. For example:

```
10      DECLARE STRING CONSTANT FM = "****.##-"
20      PRINT USING FM, 27.95
30      PRINT USING FM, -107
40      PRINT USING FM, -1007.5
50      END

RUNNH

**27.95
*107.00-
1007.50-
```

BASIC signals "PRINT USING format error" (ERR=116) if you try to print a negative number in an asterisk-fill field that does not include a trailing minus sign.

Note that you cannot specify both asterisk-fill and zero-fill for the same numeric field.

8.2.3.3 Currency Symbols

To print a currency symbol before the first digit of a number, start the field with two dollar signs. If the data contains both positive and negative numbers, end the field with a trailing minus sign:

```
10       DECLARE STRING CONSTANT FM = "$$##.##-"
20       PRINT USING FM; 77.44
30       PRINT USING FM; 304.55
40       PRINT USING FM; 2211.42
50       PRINT USING FM; -125.6
60       PRINT USING FM; 127.82
70       END

RUNNH

 $77.44
$304.55
% 2211.42
$125.60-
$127.82
```

Note that $$ reserves places for the currency symbol and only one digit; the $ is always printed. (Hence the warning indicator (%) when line 40 executes.) Contrast this with the asterisk-fill field, where BASIC prints asterisks only when there are leading spaces.

Note

By default, the currency symbol is a dollar sign. On VAX/VMS systems, you can change the currency symbol, radix point, and digit separator by assigning the logical names SYS$CURRENCY, SYS$RADIX, and SYS$ DIGIT_SEP.

BASIC signals "PRINT USING Format error" (ERR=116) if you try to print a negative number in a dollar sign field that does not include either a trailing minus sign or the CR/DR formatting character.

8.2.3.4 Negative Fields

To allow for a field containing negative values, specify a trailing minus sign in the format field. A negative format field causes the value to be printed with a trailing minus sign. Note that you can also denote negative fields with CR and DR. See Section 8.2.3.8.

You must use a trailing minus or the CR/DR formatting character to indicate a negative number in an asterisk fill or floating dollar sign field.

For fields with trailing minus signs, BASIC prints a minus sign after negative numbers and a space after positive numbers:

Standard Field

```
10 PRINT USING "###.##";-10.54
20 PRINT USING "###.##";10.54
30 END

RUNNH

-10.54
 10.54
```

Fields with Trailing Minus Signs

```
10 PRINT USING "##.##-";-10.54
20 PRINT USING "##.##-";10.54
30 END

RUNNH

10.54-
10.54
```

8.2.3.5 E (Exponential) Format

To print a number in E format, place four carets (^^^^) at the end of the field. The carets reserve space for:

- The capital letter E
- A plus or minus sign (which indicates a positive or negative exponent)
- A 2–digit exponent

In exponential format, BASIC does not pad the digits to the left of the decimal point. Instead, the most significant digit shifts to the leftmost place of the format field, and the exponent compensates for this adjustment:

```
10      PRINT USING "###.##^^^^",5
20      PRINT USING "###.##^^^^",1000
30      PRINT USING ".##^^^^",5
40      END

RUNNH

 500.00E-02
 100.00E+01
.50E+01
```

If you use fewer than four carets, the number does not print in E format; the carets print as literal characters. If you use more than four carets, BASIC prints the number in E format and includes the extra carets as a string literal:

```
10 PRINT USING "###.##^^^",5
20 PRINT USING "###.##^^^^^",5
30 END

RUNNH

   5.00^^^
 500.00E-02^
```

You must reserve a place for a minus sign to the left of the decimal point to display negative numbers in exponential format. If you do not, BASIC signals a warning error.

You cannot use exponential format with asterisk-fill, floating dollar sign, or trailing minus formats.

8.2.3.6 Leading Zeros

To print leading zeros in a numeric field, start the format field with a zero (0) enclosed in angle brackets. These characters also reserve one place for a digit. For example:

```
100     FM$ = "<0>####.##"
200     PRINT USING FM$, 1.23, 12.34, 123.45, 1234.56, 12345.67

RUNNH

00001.23
00012.34
00123.45
01234.56
12345.67
```

When you specify zero-fill, you cannot specify asterisk-fill nor floating dollar sign format for the same field.

8.2.3.7 Blank-If-Zero Fields

To make BASIC print a blank field for values which round to zero, start the numeric field with a percent sign (%) enclosed in angle brackets. For example:

```
100      FM$ = "<%>#####.##"
200      PRINT USING FM$, 1000, 0, .001, -5000

RUNNH

  1000.00

 -5000
```

PRINT USING displays spaces in each reserved position for the second and third items in the list. The value of the second item is zero, while the value of the third item becomes zero when rounded to fit the numeric field.

8.2.3.8 Debits and Credits

You can have BASIC use credit and debit notation to differentiate positive and negative numbers. To do this, you place <CD> (Credit/Debit) at the end of the numeric format string. This causes BASIC to print CR (Credit Record) after negative numbers and zero, and DR (Debit Record) after positive numbers. For example:

```
100      FM$ = "$$####.##<CD>"
200      PRINT USING FM$, -552.35, 200, -5

RUNNH

 $552.35CR
 $200.00DR
   $5.00CR
```

Note that you cannot use trailing minus sign and Credit/Debit formatting in the same numeric field.

8.3 Printing Strings

The PRINT USING statement lets you format strings by specifying:

- The number of characters
- Left-justified format
- Right-justified format
- Centered format
- Extended field format

Table 8–2 summarizes the format characters and their effects.

Table 8–2: Format Characters for String Fields

Character	Effect on Format
' single quotation mark	Starts the string field and reserves place for one character.
L (upper- or lowercase)	Left-justifies the string and reserves place for one character.
R (upper- or lowercase)	Right-justifies the string and reserves place for one character.
C (upper- or lowercase)	Centers the string in the field and reserves place for one character.
E (upper- or lowercase)	Left-justifies the string; expands the field, as necessary, to print the entire string; reserves place for one character.
\ \ two backslashes	Reserves n + 2 character positions, where n is the number of spaces between the two backslashes. PRINT USING left-justifies the string in this field. This formatting character is included for compatibility with BASIC–PLUS. DIGITAL recommends that you do not use this type of field for new program development.
! exclamation point	Creates a 1–character field. The exclamation point both starts and ends the field. This formatting character is included for compatibility with BASIC–PLUS. DIGITAL recommends that you do not use this type of field for new program development. Instead, use a single quotation mark to create a 1–character field.

String format fields start with a single quotation mark (') that reserves a space in the print field, followed by:

- A contiguous series of upper- or lowercase Ls for left-justified output
- A contiguous series of upper- or lowercase Rs for right-justified output
- A contiguous series of upper- or lowercase Cs for centered output
- A contiguous series of upper- or lowercase Es for extended field output

BASIC ignores the overflow of strings larger than the string format field except for extended fields. For extended fields, BASIC extends the field to print the entire string. If a string to be printed is shorter than the format field, BASIC pads the string field with spaces.

A string field containing only a single quotation mark is a 1–character string field. BASIC prints the first character of the string expression corresponding to a 1–character string field and ignores all following characters:

```
10 PRINT USING "'","ABCDE"
20 END

RUNNH

A
```

8.3.1 Left-Justified Format

BASIC prints strings in a left-justified field starting with the leftmost character. BASIC pads shorter strings with spaces and truncates longer strings on the right to fit the field.

A left-justified field contains a single quotation mark followed by a series of Ls:

```
10 PRINT USING "'LLLLLL",ABCD"
20 PRINT USING "'LLLL",ABC"
30 PRINT USING "'LLLLL","12345678"
40 END

RUNNH

ABCD
ABC
123456
```

8.3.2 Right-Justified Format

BASIC prints strings in a right-justified field starting with the rightmost character. BASIC pads the left side of shorter strings with spaces. If a string is longer than the field, BASIC left-justifies and truncates the right side of the string.

A right-justified field contains a single quote mark (') followed by a series of Rs:

```
5       DECLARE STRING CONSTANT RIGHT_JUSTIFY = "'RRRRR"
10      PRINT USING RIGHT_JUSTIFY,"ABCD"
20      PRINT USING RIGHT_JUSTIFY,"A"
30      PRINT USING RIGHT_JUSTIFY,"STUVWXYZ"
40      END

RUNNH

  ABCD
     A
STUVWX
```

8.3.3 Centered Fields

BASIC prints strings in a centered field with the center of the string in the center of the field. If BASIC cannot exactly center the string — as is the case for a 2–character string in a 5–character field, for example — BASIC prints the string one character off center to the left.

A centered field contains a single quotation mark followed by a series of Cs:

```
10      DECLARE STRING CONSTANT CENTER = "'CCCC"
20      PRINT USING CENTER, "A"
30      PRINT USING CENTER, "AB"
40      PRINT USING CENTER, "ABC"
50      PRINT USING CENTER, "ABCD"
60      PRINT USING CENTER, "ABCDE"
70      END

RUNNH

   A
  AB
  ABC
 ABCD
 ABCDE
```

If there are more characters than places in the field, BASIC left-justifies and truncates the string on the right.

8.3.4 Extended Fields

An extended field contains a single quotation mark followed by one or more Es. The extended field is the only field that automatically prints the entire string. In addition:

- If the string is smaller than the format field, BASIC left-justifies the string as in a left-justified field.
- If the string is longer than the format field, BASIC extends the field and prints the entire string.

For example:

```
10 PRINT USING "'E", "THE QUICK BROWN"
20 PRINT USING "'EEEEEEE', "FOX"
30 END

RUNNH

THE QUICK BROWN
FOX
```

This example uses extended, left-justified, right-justified, and centered fields.

```
10      PRINT USING "'LLLLLLLLL","THIS TEXT"
20      PRINT USING "'LLLLLLLLLLLLL","SHOULD PRINT"
30      PRINT USING "'LLLLLLLLLLLLL",'AT LEFT MARGIN'
40      PRINT USING "'RRRR","1,2,3,4"
50      PRINT USING "'RRRR",'1,2,3'
60      PRINT USING "'RRRR","1,2"
70      PRINT USING "'RRRR","1"
80      PRINT USING "'CCCCCCCCC","A"
90      PRINT USING "'CCCCCCCCC","ABC"
100     PRINT USING "'CCCCCCCCC","ABCDE"
110     PRINT USING "'CCCCCCCCC","ABCDEFG"
120     PRINT USING "'CCCCCCCCC","ABCDEFGHI"
130     PRINT USING "'LLLLLLLLLLLLLLLLL","YOU ONLY SEE PART OF THIS"
140     PRINT USING "'E","YOU CAN SEE ALL OF THE LINE WHEN EXTENDED"
150     END

RUNNH

THIS TEXT
SHOULD PRINT
AT LEFT MARGIN
1,2,3
1,2,3
  1,2
    1
    A
   ABC
  ABCDE
 ABCDEFG
ABCDEFGHI
YOU ONLY SEE PART
YOU CAN SEE ALL OF THE LINE WHEN EXTENDED
```

8.4 PRINT USING Statement Error Conditions

There are two types of PRINT USING error conditions: fatal and warning. BASIC prints warning messages in the format (%text) and continues execution. However, warnings mean that the output might not be in the intended format.

BASIC signals a fatal error if:

- The format string is not a valid string expression
- There are no valid fields in the format string
- You specify a string for a numeric field
- You specify a number for a string field
- You separate the items to be printed with characters other than commas or semicolons
- A format field contains an invalid combination of characters
- You print a negative number in a floating dollar sign or asterisk-fill field without a trailing minus sign

BASIC issues a warning if a number does not fit in the field. If a number is larger than the field allows, BASIC prints a percent sign (%) followed by the number in the standard PRINT format.

If a string is larger than any field other than an extended field, BASIC truncates the string and does not print the excess characters.

If a field contains an invalid combination of characters, BASIC does not recognize the first invalid character or any character to its right as part of the field. They may form another valid field or be considered text. If the invalid characters form a new valid field, this, like any field, can cause a fatal error condition if the item to be printed does not match the field.

Consider the following examples of invalid character combinations in numeric fields.

1.
```
10 PRINT USING "$$*****.##",5.41,16.30
```

   Explanation:

   $$ forms a complete field and **##.## forms a second valid field. The first number (5.41) is formatted by the first valid field ($$). It prints as "$5". The second number (16.30) is formatted by the second field (**##.##) and prints as "**16.30".

   The output is:

```
$5**16.30
```

2.
```
10 PRINT USING "##.#^^^^",5.43E09
```

   Explanation:

   The field has only three carets instead of four. BASIC prints a percent sign and the number, followed by the ^^^.

   The output is:

```
% .543E+10^^^
```

3.
```
10 PRINT USING "'LLEEE","VWXYZ"
```

   Explanation:

   You cannot combine two letters in one field. BASIC interprets EEE as a string literal.

   The output is:

```
VWXEEE
```

Note

Any future formatting characters will be implemented with the special character enclosed in angle brackets. Therefore, you should not include angle brackets as literals in a format string unless they are preceded by an underscore.

Chapter 9
RMS Files

This chapter explains the BASIC file organizations and record operations that are implemented through DIGITAL's Record Management Services (RMS–11 on PDP–11 systems and VAX–11 RMS on VAX/VMS systems). For a thorough understanding of file organization and file and record operations, see the RMS user's guide for your system.

BASIC also supports system-specific I/O such as native-mode files on RSTS/E systems, mailbox I/O on VAX/VMS systems, and I/O to hardware devices. See *BASIC on VAX/VMS Systems*, *BASIC on RSTS/E Systems*, or *BASIC on RSX–11M/M–PLUS Systems* for details.

RMS stores data in physical records, or blocks. A block is the smallest number of bytes BASIC transfers in a read or write operation. On disk, a block is 512 bytes. On magnetic tape, it is between 18 and 8192 bytes.

RMS stores one or more data records in each block. A data record may also be divided into smaller units, called fields. A data record can be smaller than, equal to, or larger than a disk block.

From your program's point of view, a data record is the information contained in a record buffer. A record buffer is an area in the program's storage that is treated as a unit. You can declare static record buffers (with MAP statements) or dynamic record buffers (with MOVE statements). You can redefine static record buffers with the REMAP statement.

Before reading further in this chapter, you should become familiar with MAP, MAP DYNAMIC, REMAP, and MOVE statements. See Chapter 5 for an explanation of MAP, MAP DYNAMIC, and REMAP statements. See Section 9.6.2.3 for information about MOVE statements.

9.1 Record Formats

The record format determines how RMS stores a record in the block. You specify record format in the OPEN statement. You can select either fixed-length or variable-length records.

Fixed-length records are all the same length. RMS stores fixed-length records as they appear in the record buffer, including any spaces or null characters following the data in the record buffer (this is called padding). Processing these records involves less overhead than other record formats. However, this format can use disk storage space less efficiently than variable-length records.

Variable-length records can have different lengths, but no record can exceed a maximum size you set for the file. When the record is written to a file, RMS adds a 2–byte binary record length header to each record. This count gives the length of the record (excluding the header) in bytes. When your program retrieves a record, this header is not included in the record buffer.

While variable-length records usually make more efficient use of storage space, manipulation of the record length headers generates processor overhead.

Specifying variable-length format for relative files does not save disk space. RMS always uses a fixed-length cell for each record in the file and fills any remaining space in the cell with null characters. However, specifying variable-length will prevent the padding from appearing when you access a record.

9.2 File Organizations

RMS provides sequential, relative, indexed, and block I/O files. If you specify no organization clause when creating the file, the default is a terminal-format file.

Note

On RSTS/E systems, only files opened with ORGANIZATION SEQUENTIAL, RELATIVE, or INDEXED are RMS files. Files opened with ORGANIZATION VIRTUAL or with no ORGANIZATION clause are RSTS/E native-mode files. See *BASIC on RSTS/E Systems* for more information.

- Sequential

  A sequential file contains records stored in the order that they were written. Your program specifies the record format (fixed- or variable-length), and accesses data with GET and PUT statements. You read a sequential file from beginning to end; therefore sequential files are most useful when you access the data in the file sequentially each time you use it. Sequential files can reside on both disk and magnetic tape devices.

- Relative

  A relative file contains a series of cells. Each cell can contain only a single record. For fixed-length records, the length of each cell equals the record length plus one byte. For variable-length records, the length of the cell equals the maximum record size plus three bytes. Your program specifies: 1) the record format and 2) the divisions or fields within the record. You access data with GETs and PUTs, either randomly by cell number or sequentially by omitting cell numbers. Relative files reside on disk devices only.

  Relative files are most useful when: 1) randomly accessed and 2) record contents correspond to cell numbers (for example, when inventory numbers correspond to cell numbers).

- Indexed

  Indexed files contain data records sorted in ascending order by primary index key value. An index key is a record field by which the records are ordered. For example, the primary index key for personnel records might be the employee number. Indexed files must have a primary index key and can also contain one or more alternate index keys. Your program specifies the format of data records and accesses the data with GET and PUT statements. Because records are ordered by key values, the data can be accessed randomly by specific key value, or sequentially according to ascending key value. Indexed files reside on disk devices only.

Indexed files are most useful when: 1) randomly accessed and 2) you may want to access the records in more than one way. You can specify index keys to access a file by one of several record fields.

- Block I/O

  Block I/O files are random access files that store one or more data records in each 512–byte disk block. You specify the location and format of the data by blocking and deblocking records with MAP, REMAP, and MOVE statements. Block I/O files are more efficient than other types of files because they do not use RMS buffering. All data transfers are between the file and the program's record buffer.

  You open the file with ORGANIZATION VIRTUAL and access data with GET and PUT statements, either randomly by block number or sequentially by omitting the block numbers. Each GET or PUT moves an integral number of 512–byte data blocks.

- Terminal-Format

  On all systems except RSTS/E, terminal format files are RMS sequential files of variable-length records. Each record is less than or equal to the specified record size. You create a terminal-format file by omitting the ORGANIZATION clause in the OPEN statement. You write records with the PRINT # and PRINT # USING statements, and read them with INPUT #, INPUT LINE #, and LINPUT # statements. See Chapter 2 for more information about simple I/O to terminal-format files. You can also use MAT statements to move data between terminal-format files and arrays. See Chapter 7. On RSTS/E systems, terminal-format files are RSTS/E native mode stream files. See *BASIC on RSTS/E Systems* for more information.

- Virtual Arrays

  On PDP–11 systems, virtual array files let you use disk files for arrays too big to fit in memory. You open virtual array files with ORGANIZATION VIRTUAL and dimension an array on the file with the DIM # statement. You perform operations on virtual arrays just as you do on in-memory arrays. Virtual arrays can contain integer, floating-point, or string data.

  Because VAX/VMS is a virtual memory system, conserving memory is not as important a consideration. Virtual arrays are supported for compatibility with PDP–11 BASIC–PLUS–2.

- Undefined

  Undefined files are files with unknown characteristics. You open them FOR INPUT only. Specifying UNDEFINED causes BASIC to ignore the file attribute checking normally done on files. After you have opened the file, you use the STATUS keyword to find out what the file attributes are. This usage should be reserved for special applications only.

9.3 Record Access and Record Context

Record access modes affect the order in which your program retrieves records from a file. They determine the record context, that is, the Current Record and the Next Record to be processed.

RMS allows three methods of record access:

- Sequential
- Random by key
- Random by RFA (Record File Address)

Sequential access is valid on files of any organization. Random-by-key access is valid on relative and indexed files. Random-by–RFA access is valid on any RMS file.

Thus, it is possible to access relative and indexed files sequentially. This means that your program can open a relative or indexed file and perform a series of GET statements that do not specify a record number or key value. Each GET statement retrieves the next logical record.

In the case of relative files, the next logical record is that with the next higher record number (or cell number). In the case of indexed files, the next logical record is that with the next higher value in the current key of reference.

You can also access relative and indexed files randomly by key. This means that a GET statement specifies a particular record to be retrieved. For RELATIVE files, the key specification is the record (or cell) number. For indexed files, the key specification is a record (a primary or alternate key) and an integer or string value.

You can access 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. The RFA 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 Section 9.6.2.5 for more detail.

Your program can change access modes while reading and writing records. For example, if a relative file contains both master and detail records for each employee, you might randomly access the master record for a particular employee, then perform sequential GETs to retrieve the associated detail records.

After your program opens a file, you can use the following statements to perform I/O:

- GET
- PUT
- FIND
- DELETE
- UPDATE
- SCRATCH
- FREE
- UNLOCK
- RESTORE

The record context is the state of the Current Record Pointer and Next Record Pointer. When your program successfully executes one of the listed statements, the record context can change. If an I/O statement is unsuccessful, the record pointers do not change. Table 9–1 shows how each statement affects record context.

Note that successfully executing a GET statement may or may not change the record context. If the last I/O operation before the GET was a FIND, GET operates on the Current Record. Otherwise, GET operates on the Next Record. This behavior improves program performance for the following reason: FIND statements execute faster than GET statements because FIND does not move any data; it merely locates the specified record and makes it the Current Record. Thus, your program can move through an indexed file with FIND statements, testing each record's key. When the desired record has been located, a GET statement actually moves the Current Record from the file into the record buffer.

Table 9–1: I/O Statements and Record Context

Record Operation	Record Access Mode	Current Record	Next Record
GET (Last operation not a FIND)	Sequential	New	New Current Record + 1
GET (Last operation was a FIND)	Sequential	Unchanged	Current Record + 1
GET	Random	New	New Current Record + 1
PUT	Sequential	None	1. Sequential file — end of file 2. Relative file — next record position 3. Indexed file — undefined
PUT	Random	None	Unchanged
FIND	Sequential	New	New Current Record + 1
FIND	Random	New	Unchanged
UPDATE	N/A	None	Unchanged
DELETE	N/A	None	Unchanged
SCRATCH	N/A	None	End of file
UNLOCK	N/A	None	Unchanged
FREE	N/A	None	Unchanged
RESTORE #	N/A	None	First Logical Record

Notes:

1. Except for SCRATCH, RMS establishes the Current Record before establishing the identity of the Next Record.
2. The notation "+1" indicates the next sequential record as determined by the file organization. For indexed files, the current key of reference is part of the determination of the next sequential record.
3. When you execute the RESTORE # statement, the Next Record Pointer points to the first logical record in the file. For sequential and relative files, this is the first record in the file. For indexed files, it is the first record in the specified key of reference.

The alternate behavior is as follows: If the last I/O operation before a GET statement was not a FIND, GET operates on the Next Record. This lets you perform a series of GET statements (for example, in a loop), with each GET retrieving a new record.

When you first open a file, the Current Record is undefined, and (with one exception) the Next Record is always the first logical record, namely:

1. The first record in a sequential file
2. The record with the lowest cell number for a relative file
3. The first record in the current key of reference for an index file

The exception to this rule occurs when opening a sequential file with ACCESS APPEND. In this case, the Next Record is also undefined.

9.4 I/O and Record Buffers

The I/O buffer is an area in your program that is used by RMS to store data for I/O operations. You do not have direct access to the I/O buffer because it is controlled entirely by RMS. The I/O buffer holds the minimum amount of data that can be transferred to or from a device, and its size is always greater than or equal to that of the record buffer. For more information on the amount of storage allocated for the I/O buffer, see the RMS user's guide for your system.

A record buffer is also a storage area in your program; the difference between the I/O buffer and the program's record buffer is that you have direct access to and control of the record buffer. When your program reads a record from a file, the record is transferred from the file to the I/O buffer and then to the record buffer. When your program writes a record, data is transferred from the record buffer to the I/O buffer and then to the file.

Normally, you need not be concerned with the allocation of the I/O buffer; this "double-buffering" is transparent to your program. However, the I/O buffer does increase the size of your program and can be manipulated to optimize I/O performance.

MAP statements let you create *static* record buffers and associate program variables with areas (fields) of this buffer. *Static* means the size of the record buffer does not change during program execution and that program variables are always associated with the same fields in the buffer.

There are two ways to create *dynamic* record buffers. After specifying a MAP, you can use MAP DYNAMIC and REMAP statements to associate a particular program variable with a different area of the record buffer. However, the total size of a record buffer does not change during program execution.

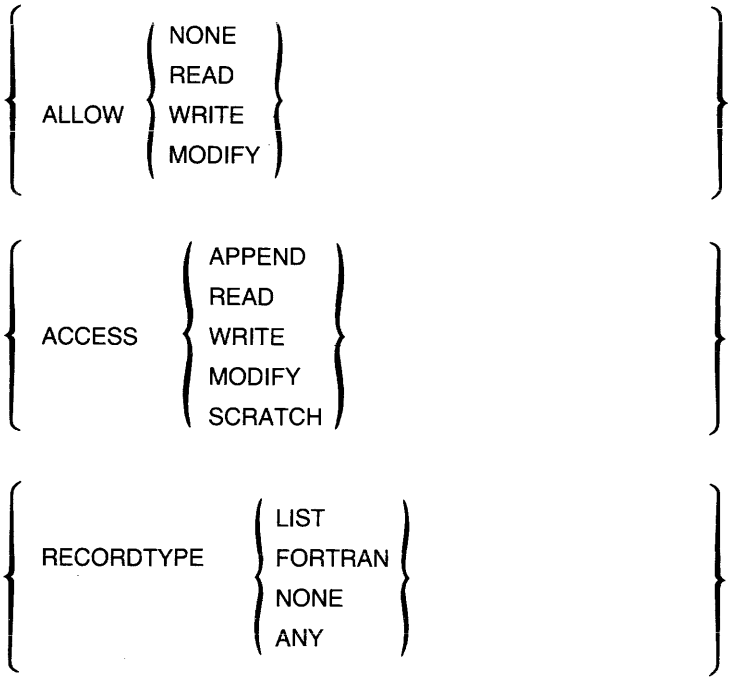
If you do not specify a MAP in the OPEN statement, the size of the record buffer is determined by the OPEN statement RECORDSIZE clause, or the record size associated with the file. If you don't specify a MAP, you must use MOVE TO and MOVE FROM statements to transfer data back and forth from the record buffer to program variables. MOVE statements do not transfer data to or from a file.

Note that you can specify both a MAP clause and a RECORDSIZE clause in the OPEN statement. In this case, the specified RECORDSIZE must be less than or equal to the size of the MAP and overrides the size specified by the MAP clause.

9.5 Opening Files (OPEN Statement)

The OPEN statement opens a file for processing, specifies the characteristics of the file to RMS, and verifies the result. Its format is:

```
                    [ FOR INPUT  ]
OPEN file-spec1     [ FOR OUTPUT ]   AS [ FILE ] chnl-exp1 [, open-clause ]...

open-clause:  { [ ORGANIZATION ]  { VIRTUAL    }   [ STREAM   ] }
              {                   { UNDEFINED  }   [ VARIABLE ] }
              {                   { INDEXED    }   [ FIXED    ] }
              {                   { SEQUENTIAL }                }
              {                   { RELATIVE   }                }
```



{ 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 sub-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)

(continued on next page)

```
{   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-clause:  { str-unsubs-vbl
             { int-unsubs-vbl
             { decimal-unsubs-vbl
             { ( str-unsubs-vbl1 ,... str-unsubs-vbl8 )
```

where:

file-spec Is the file specification. File-spec must be a valid file specification on your system. It can be a string constant, literal, or variable.

chnl-exp Is a channel number associated with the file for as long as it is open. The channel number identifies the file in I/O statements such as GET, PUT, and so on.

Note that BASIC–PLUS–2 on RSTS/E systems also allows a CLUSTERSIZE clause and a STREAM record format. See *BASIC on RSTS/E Systems* for more information.

Opening a file FOR INPUT specifies that you want to use an existing file. Opening a file FOR OUTPUT specifies that you want to create a new file. If you specify neither FOR INPUT nor FOR OUTPUT, BASIC tries to open an existing file; if no such file exists, BASIC creates a new file.

The following sections show examples of using the OPEN statement on sequential, relative, indexed, and block I/O (virtual) files. See the *BASIC Reference Manual* for complete explanations of each of the OPEN clauses. To perform I/O to devices and I/O optimization, see the user's guide for your system.

9.5.1 Opening Sequential Files

The following program opens a sequential file and associates it with a static record buffer. The ACCESS APPEND clause sets the record pointers to the end of the file:

```
20      MAP (MAP1) STRING NA.ME = 32%,                  &
                LONG DEPT.NUM,                          &
        DOUBLE BUDGET
30      OPEN "CASE.DAT" FOR INPUT AS FILE #5%           &
                ,ORGANIZATION SEQUENTIAL FIXED          &
                ,ACCESS APPEND, ALLOW NONE              &
                ,MAP MAP1
```

The MAP statement in line 20 defines the record buffer's total size and the data types of its variables. The MAP clause in line 30 references this MAP and associates it with the file on channel number 5. The data record contains one 32–byte string, one LONG integer, and one double-precision floating-point number. The record size is the total of these fields, or 44 bytes. BASIC statically allocates a record buffer 44 bytes long; all record operations use this buffer for I/O to the file.

The following program opens a sequential file for reading only (ACCESS READ), with a dynamic record buffer:

```
10      OPEN "CASE.DAT" AS FILE #1%/n                    &
            ,ORGANIZATION SEQUENTIAL VARIABLE            &
            ,ACCESS READ                                 &
            ,RECORDSIZE 100%
```

Because the OPEN statement contains no MAP clause, BASIC allocates an area for the record buffer out of the program's free space. This dynamic record buffer is 100 bytes long. Your program must use MOVE TO and MOVE FROM statements to move data between the record buffer and a list of variables.

9.5.2 Opening Relative Files

This program opens a relative file and uses static buffering:

```
110     MAP (TEST) LONG PART_NUMBER,                      &
                STRING INV_NAME = 16,                    &
                DOUBLE UNIT_PRICE
120     OPEN "RELATV.FIL" FOR OUTPUT AS FILE #1%      &
            ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY, &
            ,ALLOW READ, MAP TEST
```

The MAP statement in line 110 defines the record buffer's total size and the data types of its variables. When the program is compiled, BASIC allocates space in the record buffer for one integer, one 16–byte string (the default if you do not specify a length), and one floating-point number. The record size is the total of these fields or 24 bytes. All record operations use this buffer for I/O to the file.

This program opens a relative file with a dynamic record buffer:

```
110     OPEN "RELATV.FIL" FOR OUTPUT AS FILE #1%  &
            ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY, &
            ,ALLOW READ, RECORDSIZE 220%
```

Your program must use MOVE TO and MOVE FROM statements to move data between the record buffer and a list of variables.

9.5.3 Opening Indexed Files

Indexed files let you store records in a predefined sorted order. The keys (record fields) you specify determine the order in which the records are logically stored. Keys must be variables declared in a MAP statement; they can be: 1) strings, 2) WORD integers, 3) LONG integers, or 4) packed decimal numbers (VAX–11 BASIC only). String keys can also be *segmented*, that is, the key can be composed of up to eight string variables in the MAP.

When you create an indexed file, you must specify a primary key, and you can specify up to 254 additional alternate keys. RMS creates one index for each 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. Thus, it does not matter where the physical records are located; multiple keys mean only that there are multiple lists of record pointers, each sorted in a different order.

The OPEN statement for indexed files must have a MAP clause (which references a MAP statement) and, if you are creating the file, a PRIMARY KEY clause. The MAP statement defines the record buffer and associates program variables with parts of that buffer. The MAP statement should lexically precede the OPEN statement that references it.

The PRIMARY KEY clause names one of the mapped variables as the field by which the file is sorted. In addition, the OPEN statement can contain ALTERNATE KEY clauses that correspond to alternate access paths. When creating an indexed file, you must specify a primary key. For example:

```
10      MAP (REC) STRING RECNO,                          &
            LONG INV_NUM,                                &
            STRING PROJECT_NUM = 8%,                    &
            STRING JOB_SUPERVISOR = 32%

20      OPEN "INDEX.DAT" FOR OUTPUT AS FILE #8%          &
            ,ORGANIZATION INDEXED FIXED                   &
            ,PRIMARY KEY RECNO                             &
            ,ALTERNATE KEY INV_NUM                         &
            ,ALTERNATE KEY PROJECT_NUM                    &
            ,ALTERNATE KEY JOB_SUPERVISOR                 &
            ,MAP REC
```

This OPEN statement specifies four index keys. When this file is created, RMS includes four record pointer lists, each one ordered by the values of its associated key variable.

You can also create an index key containing more than one string variable by separating the variables with commas and enclosing them in parentheses. All the string variables must be part of the associated MAP. For example:

```
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
```

In this example, the primary key is made up of three string variables.

By default, BASIC assumes that key values are unique for each record and that no changes can be made to key values. To allow duplicate key values or changes to key values, you must specify those attributes in the OPEN statement. For example.

```
,PRIMARY KEY RECNO DUPLICATES
,ALTERNATE KEY JOB_SUPERVISOR DUPLICATES CHANGES
```

You cannot specify CHANGES for the primary key. If you do not allow duplicates on a particular key and attempt to insert a record whose key values match one already in the file, BASIC signals "Duplicate key detected" (ERR = 134). Also, if you do not allow changes on an alternate key and that key value would be changed by the execution of an UPDATE statement, BASIC signals "Key not changeable" (ERR = 130).

Although you cannot create an indexed file without at least one key clause, you can open an existing indexed file without specifying a key clause.

9.5.4 Opening Block I/O Files

Block I/O files contain a series of 512–byte blocks. You open the file with a record size of 512 or multiple of 512. For any record size that is not a multiple of 512, BASIC rounds the value to the next higher multiple. For example:

```
50        MAP (VIRT) STRING V = 512%
100       OPEN "VIRT.DAT" FOR OUTPUT AS FILE #1%,       &
                VIRTUAL,                                &
                MAP VIRT
```

This OPEN statement creates a file with a fixed-length, 512–byte record size.

9.5.5 Opening Terminal-Format Files

On all systems except RSTS/E, terminal-format files are RMS sequential files with variable-length records. On RSTS/E systems, terminal-format files are RSTS/E native-mode stream files. See *BASIC on RSTS/E Systems* for more information about stream files. You create a terminal-format file by not specifying an ORGANIZATION clause in the OPEN statement. For example:

```
100     OPEN "TERM.DAT" FOR OUTPUT AS FILE  1%,     &
        RECORDSIZE 80%,                             &
        ALLOW READ
```

This OPEN statement creates an RMS sequential file with variable-length records and a record size of 80. The ALLOW clause specifies that other users can read the file.

9.5.6 Opening Undefined Files

If you do not know what a file's organization is, you can find out by opening the file for input with ORGANIZATION UNDEFINED and RECORDTYPE ANY. Your program can then use the FSP$ function to determine the characteristics of that file. Your program must execute FSP$ immediately after the OPEN FOR INPUT statement. The syntax of the FSP$ function is:

str-vbl = FSP$(channel-number)

where:

str-vbl Is a string variable.

For example:

```
10        MAP (A) A$ = 32
20        MAP (A) WORD ORG_RAT, MRS, LONG ALQ,          &
              WORD BKS_BLS, NUM_KEYS, LONG MRN
30        OPEN "FIL.DAT" FOR INPUT AS FILE #1%,         &
              ORGANIZATION UNDEFINED,                   &
              RECORDTYPE ANY, ACCESS READ
40        A$ = FSP$(1%)
```

In this program, FSP$ generates the following values:

- ORG_RAT, which returns file characteristics:
  - High Byte is the RMS organization (ORG) field.
  - Low byte is the RMS record attributes (RAT) field.
- MRS returns the RMS maximum record size (MRS) field.
- ALQ returns the RMS allocation quantity (ALQ) field.
- BKS_BLS returns the RMS bucket size (BKS) field for disk files or the RMS block size (BLS) field for magnetic tape files.
- NUM_KEYS returns the number of keys.
- MRN returns the RMS maximum record number (MRN) if the file is relative.

See your RMS user's guide for more information.

Note

On PDP–11 systems, bytes 9 and 10 return the RMS bucket size (BKS) or RMS block size (BLS) for magnetic tape. On VAX/VMS systems, FSP$ returns zeros in bytes 9 through 12. Note that you can also find this information with a user-supplied program specified in the USEROPEN clause. See Section 9.12.11 for more information about USEROPEN.

9.6 Record Operations

The following sections describe the BASIC I/O statements and the operations they perform. BASIC I/O statements are:

PUT	Moves data from the record buffer to a file.
GET	Moves data from the file to the record buffer.
FIND	Makes a specified record the Current Record for a GET, UPDATE, or DELETE operation.
DELETE	Removes a record from a file.
UPDATE	Replaces the Current Record in a file with the data in the record buffer.
SCRATCH	Truncates a sequential file. Deletes all records from the Current Record to the end of the file, inclusive.
LOCK	Prevents other users from accessing a specified record or bucket.
UNLOCK	Unlocks a specified record or bucket locked by a FIND, GET, or LOCK statement.
FREE	(VAX–11 BASIC only) Unlocks all records and buckets for a specified channel.
RESTORE #	Resets the Next Record Pointer to the first logical record. For relative and sequential files the first logical record is the first record in the file. For indexed files, the first logical record is the first record in the specified key of reference.

9.6.1 Locating Records (FIND Statement)

The FIND statement locates a specified record and makes it the Current Record. Using FIND to locate records has the following advantages:

- FIND does not transfer any data; therefore, it executes faster than a GET statement.
- If the RMS index lists are in memory, a FIND on an indexed file does not initiate any disk operations.

FIND sets the Current Record Pointer to the record just found, thus making it the target for a GET, PUT, UPDATE, or DELETE statement.

For sequential access, the FIND statement format is:

```
FIND #chnl-exp
```

where:

chnl-exp Is a channel associated with an open file.

Sequential access is valid on RMS sequential, relative, indexed and block I/O files. A successful sequential access FIND updates both the Current Record and Next Record pointers.

A sequential FIND operates in the following ways:

- For sequential files, it locates the Next Record in the file, makes this the Current Record, and changes the Next Record to the Current Record plus one.
- For relative files, it locates the record with the next higher record number (or cell number), makes this the Current Record, and changes the Next Record to the Current Record plus one.
- For indexed files, it 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, it locates the next disk block (for files with RECORDSIZE 512) or the next record (for files with RECORDSIZE greater than 512), makes this the Current Record, and changes the Next Record to the Current Record plus one.

For example:

```
100     OPEN "SEQ.DAT" FOR INPUT AS FILE #1%,      &
            ORGANIZATION SEQUENTIAL VARIABLE, &
            RECORDSIZE 132
200     FIND #1% FOR I% = 1% TO 20%
```

Line 200 performs 20 FIND operations. The twentieth record in the file is now the Current Record.

For relative and block I/O files, you can find a particular record by specifying a RECORD clause. This is called a random access FIND. A random access FIND updates the Current Record pointer but leaves the Next Record pointer unchanged. The FIND format for random access on relative and block I/O files is:

```
FIND #chnl-exp, RECORD int-exp
```

where:

chnl-exp Is a channel associated with an open relative or block I/O file.

int-exp Is an integer expression specifying the record you want to retrieve.

For example:

```
100     OPEN "BLOCK.DAT" FOR INPUT AS FILE #1%, VIRTUAL, &
            RECORDSIZE 512%
200     FIND #1%, RECORD 20%
```

This example makes the twentieth record in the file the Current Record.

For indexed files, you can locate a record by specifying a key of reference, a relational test, and a key value. The FIND format for random access on indexed files is:

FIND #chnl-exp ,KEY #int-exp1 { EQ / GE / GT } { str-exp / int-exp2 / dec-exp } (VAX–11 BASIC only)

where:

chnl-exp Is a channel associated with an open indexed file.

int-exp1 Is the key of reference. 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.

EQ
GE
GT Are relational tests that specify "equal to," "greater than or equal to," and "greater than," respectively.

str-exp
int-exp2
dec-exp Specify values to be compared with the key value of a record. This value can be a string expression, a WORD or LONG integer expression, and, in VAX–11 BASIC, a packed decimal expression.

For example:

```
100     MAP (EMP) STRING EMP_NAM, LONG EMP_NUMBER, SSN
200     OPEN "EMP.DAT" AS FILE #1%, INDEXED,        &
                ACCESS READ,                        &
                MAP EMP,                            &
                PRIMARY KEY EMP_NAM
400     FIND #5%, KEY #0% GE "JONES"
500     FIND #5%, KEY #0% GT "ABRAMSON"
```

Assume the file contains these names:

- ABELL
- ABRAMSON
- ADAMS
- HOTCHKISS
- JONES

- KNIGHT
- SMITH

Lines 400 and 500 find the records JONES and ADAMS. Line 400 identifies the key as the primary key (key number 0) and searches for the first record whose primary key is equal to or greater than JONES. JONES is the first record to meet this condition. Line 500 also uses the primary key and searches for a record whose key value is greater than ABRAMSON. ADAMS is the first record to meet this condition.

The string expression can contain fewer characters than the key of the record you hope to find. The statements on lines 400 or 500 could have specified values of fewer characters, such as "JO" in line 400 or "ABR" in line 500. Note, however, that if line 500 specified "AB", ABELL is the target record, because it is the first to meet the condition.

If you want to locate a record whose string key field exactly matches the string expression you provide, you must pad the string expression with spaces to the exact length of the key of reference. For example:

```
100     FIND #5%, KEY #0% EQ "TOM   "
200     FIND #5%, KEY #0% EQ "TOM"
```

Line 100 locates a record whose primary key field contains "TOM ". In contrast, line 200 locates the first record whose primary key field begins with "TOM".

9.6.2 Reading Records (GET Statement)

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. You should not use GET statements on terminal-format files, virtual array files, or files opened with ORGANIZATION UNDEFINED. For sequential access, the GET statement format is:

```
GET #chnl-exp
```

where:

chnl-exp Is the channel number of an open file.

Sequential access is valid on RMS sequential, relative, indexed, and block I/O files. GET retrieves the Next Record unless the last I/O statement was a FIND. In this case, GET retrieves the Current Record.

9.6.2.1 Reading Records from Sequential Files

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. For example:

```
1         ON ERROR GOTO 19000
10        MAP (DAT) STRING PROD.NAM = 30%, LONG CALL.NUM, STRING REQ.CODE
20        OPEN "RST.DAT" AS FILE #1%                    &
              ,ORGANIZATION SEQUENTIAL, MAP DAT
30      GET #1%
40      PRINT "THE PRODUCT NAME IS", PROD.NAM
50      PRINT "THE CALL NUMBER IS", CALL.NUM
60      PRINT "THE REQUISITION CODE IS", REQ.CODE
```

(continued on next page)

```
70      GO TO 30
19000   IF (ERR = 11%) AND (ERL = 30%)                  &
            THEN RESUME 19010                           &
            ELSE ON ERROR GOTO 0
19010   PRINT "END OF FILE"
19020   CLOSE #1%
32767   END
```

This example opens a sequential file and performs sequential GETs until it reaches end-of-file. At this point, the ON ERROR statement transfers control to line 19000.

9.6.2.2 Reading Records from Relative Files

Relative files allow you to retrieve records both sequentially and randomly. This example opens a relative file and performs sequential GETs until it reaches end-of-file:

```
10          ON ERROR GOTO 300
20          MAP (IMP) STRING STATE, MAIN.OFF, LONG NUM.SPL, SINGLE SCODE
30          OPEN "REC.DAT" FOR INPUT AS FILE #1%,           &
                ORGANIZATION RELATIVE,                      &
                MAP IMP, ACCESS READ
40          WHILE 1% = 1%
50          GET #1%
60              PRINT "REPORT FOR", STATE
70              PRINT "THE OFFICE FOR"; STATE; "IS", MAIN.OFF
80              PRINT MAIN.OFF; "HAS", NUM.SPL; "EMPLOYEES"
90              PRINT "THE SALES AREA CODE IS", SCODE
100         NEXT
300         IF (ERR = 11%) AND (ERL = 50%)                  &
                THEN RESUME 400 ELSE ON ERROR GOTO 0
400         PRINT "END OF FILE"
500         CLOSE #1%
600         END
```

You perform random GETs by specifying a record number. The format for a random access GET is:

GET #chnl-exp, RECORD num-exp

where:

chnl-exp Specifies an open relative file.

num-exp Specifies the record to be retrieved.

For example:

```
10          MAP (BEC) LONG VEH_NUM,                         &
                STRING SERIAL_NUM = 22%,                    &
                OWNER = 30%
20          OPEN "VEH.IDN" FOR INPUT AS FILE #2%,           &
                ORGANIZATION RELATIVE FIXED,                &
                MAP BEC, ACCESS READ
30          INPUT "WHICH RECORD DO YOU WANT";A%
40          GET #2%, RECORD A%
50          PRINT "THE VEHICLE NUMBER IS", VEH_NUM
60          PRINT "THE SERIAL NUMBER IS", SERIAL_NUM
70          PRINT "THE OWNER OF VEHICLE"; VEH_NUM; "IS", OWNER
```

(continued on next page)

```
80          INPUT "NEXT RECORD";A%
90          IF A% <> 0% THEN GOTO 40
100         CLOSE #2%
110         GOTO 32767
200         PRINT "Record Locked -- retrying"
            SLEEP 2%
            RESUME 40
32767       END
```

Random GETs set the value of the Current Record pointer to the record just read. The Next Record pointer is set to the Current Record plus one. Consider this program:

```
1000        GET #2%, RECORD 10%
1010        GET #2%
```

Line 1000 retrieves record 10, and line 1010 retrieves record 11.

Note

Record number zero and record number 1 both refer to the first record in a relative or block I/O file.

9.6.2.3 Reading Records from Block I/O Files

Block I/O files allow you to retrieve records both sequentially and randomly. 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.

Because each GET on block I/O files retrieves a record starting on a block boundary, you can reduce disk accessing by filling disk blocks completely. This also makes more records available for processing at one time.

For example, when using 128–byte records, you can store four records in each disk block. One disk access makes four records available for processing because RMS always moves at least one disk block during every disk access. In this case, the first GET causes RMS to move one 512–byte disk block into the I/O buffer. You can then access the first four 128–byte records, using the MOVE statement and FILL to deblock each record. In contrast, a file with a separate block for each record requires four times as much disk activity.

Through RMS, BASIC performs all blocking and deblocking on sequential, relative, and indexed files. However, on RMS block I/O files, your program must perform all blocking and deblocking. You can do this in one of three ways:

- With MAP statements
- With MAP, MAP DYNAMIC, and REMAP statements (dynamic mapping)
- With MOVE statements

If each logical record in the block I/O file is 512 bytes or a multiple of 512 bytes, you can use MAP statements to define the record fields. However, if the logical records are not exactly equal to a block or some integral number of blocks, you must use either dynamic mapping or MOVE statements to deblock the record.

See Chapter 5 for more information on dynamic mapping.

The MOVE statement defines data fields and moves them to and from the dynamic record buffer. The format of the MOVE statement is:

MOVE { FROM / TO } [#] chnl-exp ,vbl [,vbl]. . .

where:

FROM Moves the data from the record buffer associated with the channel number and places it in the variables in the list.

TO Moves the data from the variables in the list and places it in the record buffer associated with the channel number.

chnl-exp Is the channel number associated with the opened file.

vbl [,vbl] Is a list of the variables you move.

For example:

```
50      MOVE FROM #9%, A$, COST, NA.ME$, ID.NUM%
```

This statement moves a record with four data fields from the record buffer to the variables in the list. This includes: a string field with a default length of 16 characters (A$), a floating-point number field (COST), a second 16–character string field (NA.ME$), and an integer field (ID.NUM%).

You can input values to the variable list in your program. For example:

```
20      INPUT "NAME"; A$
30                    INPUT "COST PER UNIT"; COST
```

Valid variables in the MOVE list are:

- Scalar variables
- Arrays
- Array elements
- FILL items

You can also specify string length in the variable list. For example:

```
NA.ME$ = 30%
```

Because BASIC dynamically assigns space for string variables, the default string length during a MOVE TO is the length of the string; the default for MOVE FROM is 16 characters.

An entire array specified in a MOVE statement must include the array name, followed by an open parenthesis, followed by (n minus one) commas, where n is the number of dimensions in the array, followed by a close parenthesis. For example:

```
60          MOVE TO #5%, A$( ), C(,), D%(,,)
```

This statement moves three arrays from the program to the record buffer. A$() specifies a one-dimensional string array, C(,) specifies a two-dimensional floating-point array, and D%(,,) specifies a three-dimensional integer array.

Note

The MOVE statement moves the contents of row zero and column zero.

You specify a single array element by naming the array and the subscripts of that element, for example, A$(25) or B(3,2).

Successive MOVE statements to or from the buffer start at the beginning of the record buffer. If a MOVE TO only partially fills the buffer, the rest of the buffer is unchanged.

Thus you use the GET statement to read a record from the file. You then move the data from the buffer to variables in the list and reference the variables in your program.

A MOVE TO moves data from the variables into the I/O buffer. A PUT or UPDATE statement then moves the data from the buffer to the file.

For example:

```
5         DIM B%(3,3)
10        OPEN "MOV.DAT" AS FILE #1%  &
           ,RELATIVE VARIABLE  &
           ,ACCESS MODIFY, ALLOW NONE &
           ,RECORDSIZE 100%
20        GET #1%
          Q = Q + 1
          MOVE FROM #1%, A, B%(,), C$ = 10%
          A = A + Q
          B%(3,3) = 128%
          C$ = "NEW RECORD"
          MOVE TO #1%, A, B%(,), C$ = 10%
          UPDATE #1%
          CLOSE #1%
          END
```

This program opens file MOV.DAT, reads the first record into the buffer, modifies part of the buffer, and moves the data from the buffer into the variables specified in the MOVE FROM statement. The string length of C$ is set to 10 characters.

The MOVE TO statement moves the data from the named variables into the buffer. The UPDATE statement writes the record back into file 1 (MOV.DAT).

Note that when accessing block I/O files, you can specify a RECORD clause for the GET statement. For example:

```
1         ON ERROR GOTO 19000
10        DECLARE WORD RECORD_NUMBER
100       MAP (VIRT) STRING FILL = 512
200       MAP DYNAMIC (VIRT) STRING FIRST_NAME,      &
                    LAST_NAME,                       &
                    COMPANY
300       OPEN "VIRT.DAT" FOR INPUT AS FILE #5, VIRTUAL, MAP VIRT
350       RECORD_NUMBER = 1%
```

(continued on next page)

```
400         WHILE 1% = 1%
                GET #5, RECORD RECORD_NUMBER
                FOR I% = 0% TO 3%
                    REMAP (VIRT) STRING FILL = ( I% * 128% ),     &
                          FIRST_NAME = 20,                        &
                          LAST_NAME = 44,                         &
                          COMPANY = 64
                    PRINT FIRST_NAME, LAST_NAME, COMPANY
                NEXT I%
                RECORD_NUMBER = RECORD_NUMBER + 1%
            NEXT
19000       IF ERR = 11%
            THEN
                PRINT "Finished"
                RESUME 32767
            ELSE ON ERROR GOTO 0
            END IF
32767       END
```

This program opens a block I/O file with each physical record containing 512 bytes. However, each logical record is 128 bytes long, and consists of a 20–character first name field, a 44–character last name field, and a 64–character company name field. The RECORD_NUMBER variable is initially set to 1, and the GET statement uses this variable to retrieve the specified record. This variable is incremented within the WHILE loop so that the next 512–byte record is retrieved in each WHILE loop iteration.

The FOR/NEXT loop within the WHILE loop deblocks the 512–byte physical records into 128–byte logical records. At each iteration of the FOR/NEXT loop, the REMAP statement uses the loop variable to mask off 128–byte sections of the record. Thus, in the first FOR/NEXT loop iteration, I% equals zero; therefore, the number of bytes specified by the FILL expression also equals zero. Because of this, FIRST_NAME points to the first 20 bytes of the record buffer, LAST_NAME points to the next 44 bytes, and so on.

In the second FOR/NEXT loop iteration, I% equals one; therefore, the number of bytes specified by the fill expression equals 128. In this iteration, FIRST_NAME points to bytes 129 through 148, LAST_NAME points to bytes 149 through 192, and COMPANY points to bytes 193 to 256. Thus, each FOR/NEXT loop iteration causes the remapped variables to point to the next logical record. The PRINT statement displays this information at each iteration.

9.6.2.4 Reading Records from Indexed Files

On indexed files you can access records randomly by specifying a key of reference and a key value. The format for a random access GET is:

GET #chnl-exp ,KEY #int-exp1 { EQ / GE / GT } { str-exp / int-exp2 / dec-exp } (VAX–11 BASIC only)

where:

chnl-exp	Is a channel associated with an open indexed file.
int-exp1	Is the key of reference. The primary key is key number 0, the first alternate key is key number 1, the second alternate key is key number 2, and so on.
EQ GE GT	Are relational tests that specify "equal to," "greater than or equal to," and "greater than," respectively.

str-exp
int-exp2
dec-exp

Specify values to be compared with the key value of a record. Dec-exp is a packed decimal number. Packed decimal key values are valid only in VAX–11 BASIC.

You can read records randomly by specifying the target value as a constant:

```
560         GET #4%, KEY #0% GT "COLUMBUS"
```

You can also specify the target value as a variable to read a number of different records:

```
10          MAP (BEC) STRING OWNER = 30%,          &
                LONG VEH_NUM,                      &
                STRING SERIAL_NUM = 22%
20          OPEN "VEH.IDN" FOR INPUT AS FILE #2%,   &
                ORGANIZATION INDEXED,              &
                MAP BEC, ACCESS READ
30          INPUT "WHICH RECORD DO YOU WANT";A$
40          GET #2%, KEY #0% EQ A$
50          PRINT "THE VEHICLE NUMBER IS", VEH_NUM
60          PRINT "THE SERIAL NUMBER IS", SERIAL_NUM
70          PRINT "THE OWNER OF VEHICLE"; VEH_NUM; "IS", OWNER
80          INPUT "NEXT RECORD";A$
90          IF A$ <> "DONE" THEN GOTO 40
100         CLOSE #2%
110         END
```

Note that the OPEN statement in this example does not need a KEY clause because the file already exists.

A random GET sets the Current Record pointer to the record just read. The Next Record pointer is set to the record logically following the Current Record in the key of reference.

Indexed files also let you access records sequentially by key value. For example:

```
10          MAP (BEC) STRING OWNER = 30%, LONG VEH_NUM, &
                STRING SERIAL_NUM = 22%
20          OPEN "VEH.IDN" FOR INPUT AS FILE #2%,   &
                ORGANIZATION INDEXED, MAP BEC, ACCESS READ
30          FOR I% = 1% TO 25%
                GET #2%
                PRINT "Vehicle Number = ";VEH_NUM
                PRINT "Owner is: ";OWNER
                PRINT
            NEXT I%
```

This example opens an indexed file and displays the first 25 records in ascending primary key value order.

9.6.2.5 Accessing Records by Record File Address

A Record File Address consists of a block number within a file and an offset into that block. This information requires six bytes of storage. The Record File Address uniquely specifies a record in a file, and accessing records by Record File Address is more efficient and faster than other forms of random record access.

Because a Record File Address requires six bytes of storage, BASIC uses a special data-type keyword to denote variables that contain Record File Address information. The data-type keyword is RFA. Note that variables of data-type RFA can be used only with the I/O statements and functions that use

Record File Address information. You cannot print these variables or use them in any arithmetic operation, except in comparisons to other RFA variables using the equal to (=) or not equal to (<>) relational operators. Also, you cannot create named constants of the RFA data type.

You can assign values from one RFA variable to another, and you can use RFA variables as parameters.

Accessing a record by Record File Address requires three steps:

- Explicitly declaring the variable of data-type RFA to hold the Record File Address
- Assigning the Record File Address to the variable with the GETRFA function
- Specifying the variable in the RFA clause of a GET or FIND statement

The GETRFA function returns the RFA of the last record accessed on a channel. Its format is:

rfa-vbl = GETRFA(chnl-exp)

where:

rfa-vbl Is a variable of the RFA data type.

chnl-exp Is the channel number of an open file. Note that you cannot include a pound sign in the channel expression.

Note

You must access a record in the file with a GET, FIND, or PUT statement before using the GETRFA function. Otherwise, GETRFA returns a zero, which is an invalid RFA.

You can use the GETRFA function only after performing a GET, FIND, or PUT. For example:

```
100     DECLARE RFA R_ARRAY(99)
        DECLARE LONG I
200     MAP (XYZ) STRING A = 80
300     OPEN "TEST.DAT" FOR OUTPUT AS FILE #1,  &
                SEQUENTIAL,   &
                MAP XYZ
400     FOR I = 0 TO 99
            !,
            !,
            !,
            PUT #1
            R_ARRAY(I) = GETRFA(1)
        NEXT I
```

This program fragment declares an array of type RFA containing 100 elements. After each PUT, the RFA of that record is assigned to an element of the array. Once the Record File Address information is assigned to a program variable or array element, you can use the RFA clause on a GET or FIND statement to retrieve the record. The format for a GET or FIND with an RFA clause is:

{GET / FIND} #chnl-exp, RFA rfa-exp

where:

chnl-exp Is the channel number of an open file.

rfa-exp Is a variable or array element of type RFA.

You can use the RFA clause on GET statements for any file organization. However, the file must reside on disk. To continue the previous example:

```
100         DECLARE RFA R_ARRAY(99)
            DECLARE LONG I
200         MAP (XYZ) STRING A = 80
300         OPEN "TEST.DAT" FOR OUTPUT AS FILE #1, SEQUENTIAL, MAP XYZ
400         FOR I = 0 TO 99
                !,
                !,
                !,
                PUT #1
                R_ARRAY(I) = GETRFA(1)
            NEXT I
1000        WHILE 1% = 1%
                PRINT "Which record would you like to see";
                INPUT "(type a carriage return to exit)";REC_NUM%
                GOTO 32766 IF REC_NUM% = 0
                GET #1, RFA R_ARRAY( REC_NUM% - 1)
                PRINT A
            NEXT
```

This program randomly retrieves the records in a sequential file by using the RFAs stored in the array.

9.6.3 Writing Records (PUT Statement)

The PUT statement moves 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. For sequential access, the PUT statement format is:

```
PUT #chnl-exp
```

where:

chnl-exp Is the channel number of an open file.

Sequential access is valid on RMS sequential, relative, indexed, and block I/O files. PUT transfers the data in the record buffer to a record in the file.

For relative and block I/O files, a sequential PUT creates a record with the next higher cell number.

When you PUT to an indexed file, RMS always stores the record in key value order and updates all index keys. This means that you do not specify sequential or random PUTs.

You can specify a RECORD clause on PUTs to relative and block I/O files. However:

- RMS allocates disk space for n records, where n is the highest cell number specified in a RECORD clause. Thus, if you OPEN a relative file and PUT record number 100, RMS allocates enough disk space for 100 records.

- If you PUT a record that already exists, BASIC signals "Record already exists" (ERR=153).

9.6.3.1 Writing Records to Sequential Files

For sequential files, a sequential PUT adds a record at the end of the file. This example opens a sequential file with ACCESS APPEND:

```
1000      MAP (BUFF) STRING CODE = 4%, EXP.DATE, TYPE.DESIG = 32%
1010      OPEN "INV.DAT" FOR INPUT AS FILE #2%,       &
              ORGANIZATION SEQUENTIAL, ACCESS APPEND
1020      FOR I% = 1% TO 560%
1030          INPUT "WHAT IS THE SPECIFICATION CODE"; CODE
1040          INPUT "WHAT IS THE EXPIRATION DATE"; EXP.DATE
1050          INPUT "WHAT IS THE DESIGNATOR"; TYPE.DESIG
1060          PUT #2%
1070      NEXT I%
```

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). After a PUT operation, there is no Current Record. The Next Record pointer is set to the end-of-file.

When processing variable-length records, you can use the COUNT clause to specify the number of bytes written. For example:

```
110       PUT #3%, COUNT 60%
```

This statement writes a 60–byte record to the file opened on channel 3. Without the COUNT clause, BASIC writes a record equal to the MAP or RECORDSIZE clause. If you have not completely filled the record buffer before executing a PUT, BASIC pads the record with nulls. If the specified COUNT is less than the buffer size, the record is truncated.

9.6.3.2 Writing Records to Relative Files

For relative files, PUT writes records sequentially or randomly. For sequential PUTs, type PUT with the file's channel number:

```
90       PUT #10%
```

RMS writes the new record in the location specified by the Next Record pointer. When setting the Next Record pointer, RMS skips over occupied cells and points to the first free cell.

A sequential PUT destroys the Current Record pointer. The Next Record pointer is set to the Next Record plus one. A random PUT destroys the Current Record pointer and leaves the Next Record pointer unchanged. For example

```
310       PUT #1%
320       PUT #1%, RECORD 20%
```

The PUT in line 320 does not change the Next Record pointer set in line 310.

When processing variable-length records, you must use the COUNT clause to specify the number of bytes written to the file. Otherwise, all records will have the record size established at OPEN time. For example:

```
90        PUT #10%, RECORD 134%, COUNT 56%
```

This statement writes a 56–byte data record into record 134. Setting the record length this way prevents unwanted data (padding) from appearing at the end of the record. The RECORD clause must specify an empty cell or BASIC signals "Record already exists" (ERR=153). The number specified in the COUNT must not exceed the size in the MAP or RECORDSIZE clause or BASIC signals "Size of record invalid" (ERR = 156).

Note that RMS also allocates disk space for empty cells. Thus, if line 90 is the first PUT statement after the file is created, RMS allocates disk space for 134 records.

9.6.3.3 Writing Records to Block I/O Files

Although block I/O files are implemented through RMS, when you write a record to a block I/O file, RMS does not perform the same error checking as with relative files. For example:

```
100       OPEN "BLOCK.DAT" AS FILE #4%, VIRTUAL, RECORDSIZE 512%
200       BLANK_RECORD$ = SPACE$(512%)
300       MOVE TO #4%, BLANK_RECORD$
400       PUT #4%, RECORD 5%
```

This program writes a record containing 512 spaces to the fifth disk block in the file, regardless of whether the block already contains a record. Note that you must use a MOVE statement to fill the record buffer before executing the PUT statement.

9.6.3.4 Writing Records to Indexed Files

For indexed files, RMS stores records in order of ascending primary key value. Therefore, you do not specify a random or sequential PUT. For example:

```
20      MAP (XXX) STRING R_NUM, DEPT_NAME, PUR_DAT
30      INPUT "REQUISITION NUMBER"; R_NUM
40      INPUT "DEPARTMENT NAME"; DEPT_NAME
50      INPUT "DATE OF PURCHASE"; PUR_DAT
60      PUT #2%
```

When writing records to an existing file, the error message "Duplicate key detected" (ERR = 134) indicates that: 1) a record with a matching key field already exists, and 2) you did not allow duplicates on that key. If you allow duplicates, RMS stores the records in a first-in, first-out sequence.

For variable-length records, you can specify a COUNT value when writing the record. This prevents unwanted data left in the buffer from printing out when a program retrieves the record. You specify COUNT as part of the PUT. For example:

```
550      PUT #2%, COUNT 124%
```

If you do not specify a COUNT, the size of the record written is the length specified by the MAP or RECORDSIZE clause in the OPEN statement (RECORDSIZE overrides MAP). When writing variable-length records, you must ensure that the record is long enough to include the primary key, or else BASIC signals "Size of record invalid" (ERR = 156).

9.6.4 Deleting Records (DELETE Statement)

The DELETE statement logically removes a record from a file. After you have deleted a record you can no longer retrieve it. DELETE works with relative and indexed files only.

A successful FIND or GET must precede the DELETE operation. These operations make the target record available for deletion. For example:

```
40      FIND #1%, RECORD 67%
50      DELETE #1%
```

The FIND statement locates record 67 and the DELETE statement removes this record from the file. Because the cell itself is not deleted, you can PUT a new record after deleting an old one. For example:

```
60      PUT #1%, RECORD 67%
```

This example writes a new record in place of the old one.

There is no Current Record after a deletion. The Next Record pointer is unchanged.

This example shows the deletion of a record from an indexed file:

```
30      FIND #2%, KEY #0% EQ "421-56-9012"
40      DELETE #2%
```

These statements delete the record with the primary key value equal to "421–56–9012".

If a DELETE statement executes while there is no current record, BASIC signals "No current record" (ERR = 131). This indicates that the previous FIND or GET was unsuccessful. When deleting or updating records, include error trapping in your program to make sure you complete FINDs and GETs successfully.

9.6.5 Updating Records (UPDATE Statement)

UPDATE writes a new record at the location indicated by the Current Record Pointer. UPDATE is valid only on RMS sequential, relative, and indexed files.

The UPDATE statement operates on the Current Record. In order to successfully update a record, you must know the exact size of the record being updated. BASIC has this information after a successful GET. Therefore, you can update a record without specifying a COUNT clause after a successful GET.

Note that FIND statements update the Current Record pointer, but because FIND statements do not move any data, BASIC does not have the record size information needed for an update operation. However, you can use the UPDATE statement after a FIND, if you specify a COUNT clause. If the value in the COUNT clause does not match the actual size of the record, BASIC signals "size of record invalid".

The error message "No current record" (ERR = 131) indicates that the GET was unsuccessful. When updating records, include error trapping in your program to make sure you complete GETs successfully.

9.6.5.1 Updating Records in Sequential Files

An UPDATE operation on a sequential file is valid only when:

- The file containing the record is on disk
- The new record is the same size as the one it is replacing

- You have either executed a successful GET on the channel or specified a COUNT clause to the UPDATE statement

The following program reads the third record into the buffer and updates it with a new record field, L.NAME:

```
10      ON ERROR GOTO 19000
30      MAP (AAA) STRING L.NAME = 60%, F.NAME = 20%, RM.NUM = 8%
40      OPEN "STU.DAT" FOR INPUT AS FILE #9%, &
            ORGANIZATION SEQUENTIAL, MAP AAA
50      INPUT "LAST NAME"; SEARCH.NAME$
70      GET #9% WHILE SEARCH.NAME$ <> L.NAME
90      INPUT "ROOM NUMBER"; RM.NUM
100     UPDATE #9%
110     GOTO 50
19000   RESUME 19010
19010   CLOSE #9%
19020   PRINT "UPDATE COMPLETE"
32767   END
```

In the absence of a COUNT clause, UPDATE uses the record size set by the last valid GET on that channel. This behavior ensures that the record written out by UPDATE is the same size as the record being replaced. If you specify a COUNT clause, the value must exactly match the size of the record just retrieved.

An UPDATE operation destroys the Current Record pointer. The Next Record pointer is unchanged.

9.6.5.2 Updating Records in Relative Files

When you UPDATE a record in a relative file, the new record can be larger or smaller than the record it replaces. However, it must be smaller than the maximum record size set when the file was opened. For example, this program updates a specified record on a relative file:

```
10      MAP (UPD) STRING ENRDAT = 8%, LONG INVOC, SH.NUM, REAL COST
20      OPEN "REC.ING" FOR INPUT AS FILE #8%, &
            RELATIVE FIXED, MAP UPD
30      INPUT "WHICH RECORD TO UPDATE";A%
35      WHILE A% <= 5000%
40          GET #8%, RECORD A%
50          INPUT "REVISED COST IS";COST
60          UPDATE #8%
70          INPUT "NEXT RECORD";A%
80      NEXT
90      CLOSE #8%
100     END
```

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. For example:

```
70      UPDATE #3%, COUNT 80%
```

This statement writes a record with a length of 80 bytes. You can specify a length equal to the latest input operation with the RECOUNT variable. However, if you use RECOUNT after an INPUT LINE statement, the number of bytes transferred includes the line terminators. For example:

```
50      GET #8%, RECORD 404%
60      INPUT LINE "NEW DATA"; NEW.DATA$
70      UPDATE #8%, COUNT (RECOUNT - 2%)
```

This program writes a record exactly equal to the length of NEW.DATA$.

You can also specify a length equal to the last GET. For example:

```
50      INPUT "TYPE IN NEW NAME"; EMP.NAME$
60      GET #8%, RECORD 591%
70      UPDATE #8%, COUNT RECOUNT
```

The RECOUNT variable is set to the length of record 591. (Because the last I/O operation was from a file and not a terminal, the value of RECOUNT does not include line terminators.) The UPDATE operation writes EMP.NAME$ with a length equal to that record. For variable-length records, GETs (not FINDs) must be used to locate the record for updating because a FIND operation does not set the size of the record.

9.6.5.3 Updating Records in Indexed Files

When you UPDATE a record in an indexed file, the new record can be larger or smaller than the record it replaces. However, 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 updated record:

- Can be no longer than the maximum record size
- Must include at least the primary key field

If the new record: 1) omits one of the old record's alternate key fields or 2) changes one of them, the OPEN statement must specify CHANGES for that key field. Otherwise, BASIC signals the error "Key not changeable" (ERR = 130).

For example, this program updates a specified record on an indexed file:

```
10      MAP (UPD) STRING ENRDAT = 8%, LONG PART.NUM, SH.NUM, REAL COST
20      OPEN "REC.ING" FOR INPUT AS FILE #8%, &
            INDEXED, MAP UPD,                    &
            PRIMARY KEY PART.NUM
30      INPUT "PART NUMBER TO UPDATE";A%
40      GET #8%, KEY #0%, EQ A%
50      INPUT "REVISED COST IS";COST
60      UPDATE #8%
70      INPUT "NEXT RECORD";A%
80      IF A% <= 5000% THEN GOTO 40
90      CLOSE #8%
100     END
```

9.6.6 Restoring Files

The RESTORE # statement returns the Current Record Pointer to the beginning of the file. RESTORE does not change the file. The RESTORE # statement has the format:

```
RESTORE #chnl-exp [,KEY #num-exp]
```

where:

chnl-exp	Is the channel number of the file you want to restore.
,KEY #num-exp	Resets an RMS indexed file by key of reference. KEY number 0 is the primary key, KEY number 1 is the first alternate, and so forth. When you restore an indexed file, specify the key you want restored. The default is the current key of reference.

For example:

```
70      RESTORE #3%, KEY #2%
90      RESTORE #3%
```

Line 70 restores the file in terms of the second alternate key. Line 90 restores the file in terms of the primary key.

All RMS file organizations can use the RESTORE # statement. RESTORE without a channel number resets the data pointer for READ and DATA statements and does not affect any files.

9.7 Truncating a File (SCRATCH Statement)

The SCRATCH statement is valid only on sequential files. Although you cannot delete single records from a sequential file, you can delete all records starting with the Current Record through to the end of the file. To do this, you must first specify ACCESS SCRATCH when you open the file.

To truncate the file, you first position the Current Record pointer by executing a FIND or GET after locating the last desired record. Then you execute the SCRATCH statement:

```
10      OPEN "MMM.DAT" AS FILE #2%    &
            ,SEQUENTIAL FIXED, ACCESS SCRATCH
30      FIND #2% FOR I% = 1% TO 33%
50      SCRATCH #2%
60      CLOSE #2%
70      END
```

This program locates the thirty-third record and truncates the file beginning with that record. SCRATCH does not change the physical size of the file. You can write records with the PUT statement immediately after a SCRATCH.

9.8 Unlocking Records (FREE and UNLOCK Statements)

When you retrieve a record from an RMS file opened with ALLOW WRITE or ALLOW MODIFY, RMS protects the integrity of the file by preventing other users from accessing that record. Note that in VAX–11 RMS, only the retrieved record is locked, while in RMS–11, the entire bucket containing the record is locked. For RMS block I/O files, this means that other users cannot access the disk block retrieved by your program.

If: 1) the same file is open in another program or on another channel in the same program and 2) another user attempts to access the record or block, BASIC signals the error "Record is locked" (ERR = 154). RMS unlocks a record or bucket when:

- Your program performs another record operation
- Your program explicitly unlocks it

For example:

```
60      FREE #8%
90      UNLOCK #6%
```

FREE unlocks all previously locked records. UNLOCK unlocks only the last record accessed. FREE and UNLOCK let two or more programs use the file at the same time.

VAX–11 BASIC allows both FREE and UNLOCK; however, BASIC–PLUS–2 supports only the UNLOCK statement. In addition, VAX–11 BASIC provides the UNLOCK EXPLICIT clause to the OPEN statement for specifying that all accessed records remain locked until explicitly unlocked with the UNLOCK or FREE statement. See *BASIC on VAX/VMS Systems* for more information.

9.9 Virtual Array Files

You create a virtual array by dimensioning an array with the DIM # statement, then opening a VIRTUAL file on that channel. You access virtual arrays just as you do normal arrays. For example:

```
100      DIM #1%, LONG INT_ARRAY(10,10,10)
.
.
.
1000     OPEN "VIRT.DAT" FOR OUTPUT AS FILE #1%, VIRTUAL
.
.
.
5000     INT_ARRAY(5,5,5) = 100%
```

Line 100 dimensions a virtual array on channel 1. Line 1000 opens a virtual file to contain the array. Line 5000 assigns a value to one array element.

You cannot redimension virtual arrays with an executable DIM statement. See Chapters 4 and 7 for more information on virtual arrays.

9.10 File Operations

This section describes how to rename, close, delete, and share files within a BASIC program.

9.10.1 Renaming Files

If the protection code permits, you can change the name or directory of a file with the NAME AS statement. The format is:

```
NAME str-exp1 AS str-exp2
```

where:

str-exp1 Is the old file name.

str-exp2 Is the new file name.

For example:

```
10       NAME "MONEY.DAT" AS "ACCNTS.DAT"
```

This statement changes the name of the file named MONEY.DAT to ACCNTS.DAT. NAME AS does not change the file's protection.

You must always include an output file type because there is no default. If you use the NAME AS statement on an open file, the new name does not take effect until you close the file.

9.10.2 Closing Files and Ending I/O

All programs should close files before the program terminates. However, BASIC automatically closes files:

- While executing a CHAIN statement
- At an END statement
- When it completes the highest numbered line in the program

BASIC does not close files after executing a STOP, SUBEND, or FUNCTIONEND statement.

The CLOSE statement has the format:

```
CLOSE [#]chnl-exp [,[#]chnl-exp]. . .
```

where:

chnl-exp Is a channel number. If you do not specify a channel, BASIC issues an error.

This is an example of CLOSE:

```
10      CLOSE#1%
20      B% = 4%
30      CLOSE#2%, B%, 6% + 1%
40      CLOSE I% FOR I% = 12% TO 1% STEP -1%
```

The CLOSE statement closes files and disassociates these files and their buffers from the file numbers. If the file was a magnetic tape device, CLOSE writes trailer labels at the end of the file.

9.10.3 Deleting Files

If the file protection code permits, you can delete a file with the KILL statement. The KILL statement has the format:

```
KILL file-spec
```

where:

file-spec Is the file specification of the file you want deleted.

For example:

```
610     KILL "TEST.BP2"
```

This program deletes the file named TEST.BP2. Note that on VAX/VMS and RSX–11M/M–PLUS systems, this statement deletes only the most current version of the file. You can delete only one file at a time. Do not omit file extensions; there is no default.

You can delete a file that is currently being accessed by other users; however, the file is not deleted until all users have closed it. You cannot open or access a file after you have deleted it.

9.11 File-Related Functions

BASIC provides built-in functions for finding:

- The current cursor position (CCPOS).
- The number of bytes moved in the last I/O operation (RECOUNT).
- The file status (STATUS).
- The margin size (MAR%). MAR% is available only in VAX–11 BASIC.

The following sections describe these functions.

9.11.1 CCPOS

The CCPOS function returns the output record's current character position on a specified channel number. Its format is:

returned-integer = CCPOS(chnl-exp)

If the channel expression is zero, CCPOS returns your terminal's current cursor position. For example:

```
10      INPUT A$, B$, C$
20      PRINT A$; B$;
30      PRINT IF CCPOS(0) > 14
40      PRINT C$
50      END
```

This program accepts three string values and prints them. However, the program checks whether the cursor is past character position 14 in the output line. If it is, BASIC prints C$ on a new line.

9.11.2 RECOUNT

Input operations can transfer varying amounts of data. The system variable RECOUNT contains the number of characters (bytes) read after each input operation. Its format is:

integer-variable = RECOUNT

After an input operation from your terminal, RECOUNT contains the number of characters transferred, including the line terminator. After accessing a file record, RECOUNT contains the number of characters in the record only.

RECOUNT is reset by every input operation on any channel, including the controlling terminal. This means that INPUT, LINPUT, and INPUT LINE statements also affect RECOUNT. Therefore, if you need to use the value of RECOUNT, copy it to a different storage location before executing another input operation. RECOUNT is not properly set if an error occurs during an input operation.

RECOUNT is often used as the argument to the COUNT clause in the PUT statement. For example:

```
1000    GET #4%
        .
        .
        .
1100    PUT #5%, COUNT RECOUNT
```

This ensures that the output record on channel five is the same length as the input record on channel four.

9.11.3 STATUS

The STATUS function accesses the status word containing characteristics of the last opened file. See the *BASIC Reference Manual* for more information on the STATUS function.

9.11.4 MAR% (VAX–11 BASIC Only)

The MAR% function returns the margin width of a specified channel. Its format is:

```
integer-variable = MAR[%](chnl-exp)
```

If the channel is 0, MAR% returns the margin width of your terminal. If the channel expression specifies a file, then MAR% returns the record size of the file.

9.12 Optimizing I/O

This section explains the OPEN statement keywords that enable you to structure your file more efficiently. These keywords are: BUCKETSIZE, TEMPORARY, FILESIZE, SPAN, CONTIGUOUS, EXTENDSIZE, CONNECT, USEROPEN, WINDOWSIZE, BUFFER, and RECORDTYPE.

9.12.1 BUCKETSIZE

The BUCKETSIZE clause applies only to relative and indexed files. A bucket is a logical storage structure that RMS uses to build and maintain relative and indexed files on disk devices. A bucket contains between 1 and 31 disk blocks, inclusive. The default bucket size is one block. Although RMS defines the bucket size in terms of disk blocks, the BASIC BUCKETSIZE clause specifies the number of records a bucket contains. For example:

```
,BUCKETSIZE 12%
```

This example specifies a bucket containing 12 records. RMS allocates enough storage from your program's address space for one bucket. A GET statement transfers one record from this storage to your program's record buffer. Thus, RMS initiates an I/O operation from the disk only when a new bucket is required.

When you open an existing relative or indexed file, and specify a BUCKETSIZE other than that originally assigned to the file, BASIC signals "File attributes not matched" (ERR = 160).

Records cannot span bucket boundaries. Therefore, when you specify bucket size in your program, you must consider the size of the largest record in the file. A bucket must contain at least one record.

There are two ways to establish the number of blocks in a bucket. The first is to use the BASIC default. The second is to specify the number of records you want in each bucket. BASIC then calculates a bucket size based on that number.

The default bucket size assigned to relative and indexed files is as small as possible. A small bucket minimizes the number of records locked when a file is shared. However, it also forces more disk transfers, especially when you are accessing records sequentially.

BASIC selects a default bucket size depending on:

- The record length
- The file organization (relative or indexed)
- The record format (fixed or variable)

If you do not define the BUCKETSIZE clause in the OPEN statement, BASIC:

- Assumes that there is a minimum of one record in the bucket
- Calculates a size
- Assigns the number of blocks

If you supply a BUCKETSIZE clause and specify the number of records, BASIC uses formulas to derive the necessary number of blocks, as follows:

Fixed-length records without BUCKETSIZE specification:

```
Bnum = (1 + Rlen)/512
```

Fixed-length records with BUCKETSIZE specified:

```
Bnum = ((1 + Rlen)*Rnum)/512
```

Variable-length records without BUCKETSIZE specification:

```
Bnum = (3 + Rmax)/512
```

Variable-length records with BUCKETSIZE specified:

```
Bnum = ((3 + Rmax)*Rnum)/512
```

where:

Bnum	Is the minimum number of blocks for each bucket, rounded up to an integer.
Rlen	Is the length in bytes of the file's fixed-length records, as defined in the RECORDSIZE clause.
Rmax	Is the length in bytes of the largest variable-length record in the file, as defined in the RECORDSIZE clause.
Rnum	Is the number of records you want in each bucket, as defined in the BUCKETSIZE clause.
1	Is the byte RMS uses to flag deleted records in the file.
3	Represents the "record deleted" byte plus two bytes that indicate the count field.

Table 9–2 shows the default bucket sizes selected by BASIC when the bucket contains the default of one record.

Table 9–2: Relative File Default Bucket Size

Number of Blocks	Record Length, Fixed-Length Records	Maximum Record Size, Variable-Length Records
1	1–511	1–509
2	512–1023	510–1021
3	1024–1535	1022–1533
4	1536–2047	1534–2045
5	2048–2559	2046–2557
6	2560–3071	2558–3069
7	3072–3583	3070–3581
8	3584–4095	3582–4093
9	4096–4607	4094–4605
10	4608–5119	4606–5117
11	5120–5631	5118–5629
12	5632–6143	5630–6141
13	6144–6655	6142–6653
14	6656–7167	6654–7165
15	7168–7679	7166–7677

BASIC derives the default bucket size for indexed files from the following formulas:

Fixed-length records without BUCKETSIZE specified:

```
Bnum = (22 + Rlen)/512
```

Fixed-length records with BUCKETSIZE specified:

```
Bnum = (((7 + Rlen)*Rnum) + 15)/512
```

Variable-length records without BUCKETSIZE specified:

```
Bnum = (24 + Rmax)/512
```

Variable-length records with BUCKETSIZE specified:

```
Bnum = (((9 + Rmax)*Rnum) + 15)/512
```

where:

Bnum	Is the number of blocks for each bucket.
Rlen	Is the length in bytes of the file's fixed-length records, as defined in the RECORDSIZE clause.
Rmax	Is the length in bytes of the largest variable-length record in the file, as defined in the RECORDSIZE clause.
Rnum	Is the number of records you want in each bucket, as defined in the BUCKETSIZE clause.

22 Is the 15–byte RMS bucket overhead plus 7 bytes for the fixed-format record header length. When you define BUCKETSIZE, BASIC allocates 7 bytes to each record in the bucket and 15 bytes to the complete bucket.

24 Is the 15–byte RMS bucket overhead plus 9 bytes for the variable-format record header length. When you define BUCKETSIZE, BASIC allocates 9 bytes to each record in the bucket and 15 bytes to the complete bucket.

Table 9–3 shows the bucket sizes selected by BASIC when the number of records is undefined.

Table 9–3: Indexed File Default Bucket Size

Number of Blocks	Record Length, Fixed-Length Records	Maximum Record Size, Variable-Length Records
1	1–490	1–488
2	491–1002	489–1000
3	1003–1514	1001–1512
4	1515–2026	1513–2024
5	2027–2538	2025–2536
6	2539–3050	2537–3048
7	3051–3562	3049–3560
8	3563–4074	3561–4072
9	4075–4586	4073–4584
10	4587–5098	4585–5096
11	5099–5610	5097–5608
12	5611–6122	5609–6120
13	6123–6634	6121–6632
14	6635–7146	6633–7144
15	7147–7658	7145–7656

If a BASIC program opens an indexed file with fixed-length records of 100 bytes and a bucket size of 5, the run-time system makes these calculations when creating the file:

- 100 bytes for the data, for example, MAP (ABC) A$ = 100 + 7 bytes for the record header
- 107 bytes for the complete record
- 535 bytes for the records in a bucket (5*107 = 535) + 15 bytes for the bucket overhead
- 550 bytes specified for each bucket

RMS requires that buckets be an integral number of blocks; therefore, the bucket size for this file is two blocks instead of one. This means that each bucket can hold at least five records. RMS continues to fill the bucket with as many records as possible.

Note

When you specify a bucket size for files in your program, keep in mind the space versus speed trade-offs. A large bucket size increases file processing speed because a greater amount of data is available in memory at one time. However, it also increases the memory space needed for buffer allocation. Likewise, a small bucket size minimizes buffer requirements, but can also decrease the speed of operations.

9.12.2 TEMPORARY

If you specify TEMPORARY in the OPEN statement, BASIC deletes the file: 1) when you close it, 2) when you log out, or 3) when the program aborts. No entry is made in any directory.

9.12.3 FILESIZE

With the FILESIZE attribute, you can allocate disk space for a file when you create it. For example:

```
100     OPEN "VALUES" FOR OUTPUT AS FILE #3%, FILESIZE 50%
```

This statement allocates 50 blocks of disk space for the file "VALUES."

Preextending a file has several advantages. First, the system can create a complete directory structure for the file, instead of allocating and mapping additional disk blocks when needed. Second, you reserve the needed disk space for your application. This ensures that you will not run out of space when the program is running. Third, preextension can make some of the file's disk blocks contiguous, especially when used with the CONTIGUOUS keyword.

Preextension can be a disadvantage if it allocates disk space needed by other users, however.

9.12.4 NOSPAN

Normally, sequential files allow records to cross (span) block boundaries. If records cross block boundaries, RMS packs records into the file end-to-end throughout the file, plus space for control information and padding.

NOSPAN overrides this default, forcing records to fit in individual blocks, plus space for control information and padding. This wastes space if your records (plus RMS overhead) do not exactly fit into a block.

When block boundaries restrict records, fixed-length records must be less than 512 bytes, and variable-length records less than 510 bytes. This can waste extra bytes at the end of the file. When records span block boundaries; however, RMS can write:

- More than one record in each block (for records shorter than 512 bytes)
- A partial record in each block (for records longer than 512 bytes)
- Records end-to-end without regard to block boundaries

For example, by specifying NOSPAN, only four 120–byte records fit into a disk block. If you do not specify NOSPAN, BASIC begins writing the fifth record in the block, and continues that record in the next block. This minimizes wasted disk space and improves the file's capacity, at the expense of increased processing overhead.

9.12.5 CONTIGUOUS

A contiguous file with physically adjoining blocks minimizes disk searching and decreases file access time. Once the system knows where a contiguous file starts on the disk, it need not use as many retrieval pointers to locate the pieces of that file. Rather, it can access data by calculating the distance from the beginning of the file to the desired data. On PDP–11 systems, if there is not enough contiguous disk space, BASIC signals an error. On VAX/VMS systems, BASIC allocates as much

contiguous space as possible — this is called contiguous-best-try in VAX–11 RMS. For truly contiguous records on VAX/VMS systems, you must use the USEROPEN clause.

Opening the file with FILESIZE and CONTIGUOUS clauses preextends the file with enough contiguous space for the entire file, if the contiguous space is available.

9.12.6 EXTENDSIZE

The EXTENDSIZE attribute determines how many disk blocks RMS adds to the file when the current allocation is exhausted. You specify EXTENDSIZE as a number of blocks. For example:

```
1000      OPEN "TSK.ORN" FOR OUTPUT AS FILE #2%, &
               ORGANIZATION RELATIVE, EXTENDSIZE 128%
```

The EXTENDSIZE clause causes RMS to add 128 disk blocks whenever the current space allocation is exhausted and the file must be extended.

The value you specify: 1) must be included when you create the file, 2) cannot exceed 65,535 disk blocks, and 3) is rounded to the next cluster boundary. If you specify zero, the extension size equals the RMS default value. The EXTENDSIZE value is a permanent file attribute.

9.12.7 CONNECT

The CONNECT clause can be used only on indexed files. CONNECT lets you process different streams of records on different indexed keys without incurring all the RMS overhead of opening the same file more than once. For example, a program can read records in an indexed file sequentially by one key and randomly by another. Each stream is an independent, active series of record operations.

For example:

```
100       MAP (INDMAP)  WORD EMP_NUM,                      &
                          STRING EMP_LAST_NAME = 20,       &
                          SINGLE SALARY,                   &
                          STRING WAGE_CODE = 2

200       OPEN "IND.DAT" FOR INPUT AS FILE #1,             &
                  ORGANIZATION INDEXED,                    &
                  PRIMARY KEY EMP_NUM,                     &
                  MAP INDMAP
          !,
          !,
          !,
          OPEN "IND.DAT" FOR INPUT AS FILE #2,             &
                  ORGANIZATION INDEXED,                    &
                  PRIMARY KEY EMP_NUM,                     &
                  MAP INDMAP,                              &
                  ALTERNATE KEY EMP_LAST_NAME,             &
                  CONNECT 1
          !,
          !,
          !,
```

```
OPEN "IND.DAT" FOR INPUT AS FILE #3,              &
        ORGANIZATION INDEXED,                     &
        PRIMARY KEY EMP_NUM,                      &
        MAP INDMAP,                               &
        ALTERNATE KEY WAGE_CODE,                  &
        CONNECT 1
```

The channel on which you open the file for the first time is called the *parent*. The CONNECT clause specifies another channel on which you access the same file. These connected channels are called *children*. More than one OPEN statement can connect to the parent channel. You cannot connect to a channel that has been already been connected to another channel.

9.12.8 WINDOWSIZE

The WINDOWSIZE attribute specifies the number of block retrieval pointers in memory for the file. WINDOWSIZE is valid only in VAX–11 BASIC and in BASIC–PLUS–2 on RSX–11M/M–PLUS systems.

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 value you supply in 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.

9.12.9 BUFFER

The BUFFER keyword applies to disk files of any organization. By default, RMS allocates enough space in your task to hold one bucket (for relative and indexed files) or one disk block (for sequential files). When the program executes the first GET statement, one bucket or one block is transferred to this RMS buffer and one record is transferred to the program's record buffer. Subsequent GETs move one record from the RMS buffer to the program's record buffer. When the OPEN statement specifies a BUFFER value, RMS multiplies its buffer by the specified value. BUFFER does not affect the size of the program's record buffer.

Thus, specifying a BUFFER value of two causes RMS to allocate enough space to hold two buckets (or two disk blocks for sequential files). Multiple buffers can speed access because more data is transferred during each disk access. However, this requires more memory.

For example:

```
1000     OPEN "RTFG.DAT" FOR INPUT AS FILE #2%, &
             ORGANIZATION INDEXED, BUFFER 5
```

This OPEN statement sets up an RMS buffer big enough to hold five buckets for the file RTFG.DAT. In BASIC–PLUS–2, you can specify up to 255 buffers. In VAX–11 BASIC, you can specify up to 127 buffers as either a positive or negative number. Table 9–4 summarizes the effects of BUFFER values in VAX–11 BASIC.

Table 9–4: BUFFER Values and Allocations in VAX–11 BASIC

Value	Allocation
0 < BUFFER < 127	RMS allocates enough space for the specified number of buckets. On VAX/VMS systems, this locks those buffers in the working set of the process.
–128 < BUFFER < 0	BASIC allocates the absolute value of the specified number of buffers. On VAX/VMS systems, this does not lock those buffers into the working set of the program.
BUFFER = 0	BASIC allocates the process default for the particular file organization and device.

If there is no process default, BASIC uses the system default for the file organization and device. If the system default is zero, BASIC allocates one buffer.

9.12.10 RECORDTYPE

The RECORDTYPE attribute lets you specify record formats compatible with files created by other language processors. You can choose one of four qualifiers: LIST, FORTRAN, ANY, or NONE. The default for BASIC is LIST, which specifies carriage return format. This is standard for ASCII text files and means that carriage control is performed by the output device.

If your program accesses a file created with a FORTRAN language processor, use the FORTRAN qualifier. For example:

```
1000      OPEN "FIL.DAT" FOR INPUT AS FILE #1%, &
              ORGANIZATION SEQUENTIAL, RECORDTYPE FORTRAN
```

The FORTRAN qualifier sets the FORTRAN carriage control attribute in the RAT field in the FAB. The first byte of the record is still assumed to be the carriage control information.

If your program accesses a file created by an unknown language processor, use the ANY qualifier. For example:

```
1000      OPEN "FIL.DAT" FOR INPUT AS FILE #1%, &
              ORGANIZATION INDEXED, RECORDTYPE ANY
```

The ANY qualifier causes BASIC to ignore any attribute except a record attribute. If you create a file with the ANY qualifier, BASIC uses the default of LIST.

For compatibility with BASIC–PLUS–2 virtual files, use the NONE qualifier. NONE lets VAX–11 BASIC read virtual array files (created with ORGANIZATION VIRTUAL).

9.12.11 USEROPEN

The USEROPEN keyword specifies a user-created external procedure that BASIC executes when you open or create a file. The procedure can specify additional file OPEN parameters. For example:

```
1000      OPEN "FILE.DAT" FOR INPUT AS FILE #2%, &
              ORGANIZATION INDEXED, USEROPEN MYOPEN
```

The code in MYOPEN determines how the file FILE.DAT opens. On VAX/VMS systems, the Run-Time Library sets up six RMS control structures before calling the USEROPEN procedure. Table 9–5 summarizes these structures and their meanings.

Table 9–5: VAX–11 RMS Control Structures Set by USEROPEN

Name	Meaning
FAB	File Access Block
RAB	Record Access Block
NAM	Name Block
XAB	FHC Extended Attributes Block
ESA	Expanded Name String
RSA	Resultant Name String

On PDP–11 systems, a USEROPEN routine sets only the FAB and RAB.

A USEROPEN procedure should not alter the allocation of these structures, although it can modify the contents of many of the fields. However, you should not modify fields set by other OPEN statement keywords. For example, use RECORDSIZE to set the record length; do not use USEROPEN.

The allocation of the RMS control structures (except for RAB on VAX/VMS systems) lasts only for the duration of the OPEN statement. Therefore, your USEROPEN can retain only the RAB address for use after the OPEN.

During execution of the USEROPEN clause:

1. BASIC performs normal OPEN statement processing up to the point where it would call the RMS OPEN/CREATE and CONNECT routines.
2. BASIC passes control to the USEROPEN routine.
3. BASIC passes the address of the FAB as the first parameter, the address of the RAB as the second parameter, and, in VAX–11 BASIC, the address of the user-specified channel number as the third parameter.
4. The USEROPEN routine calls RMS OPEN/CREATE and CONNECT routines and returns the status in R0.

For example, the following OPEN statement calls the VAX–11 MACRO routine OPENFILL contained in the MARS source module OPENFILL.MAR.

```
100      MAP (REC.1) SURNAME$ = 20%, REST$ = 60%
110      OPEN "UOPNTST.IDX" FOR OUTPUT AS FILE #1%,  &
               ORGANIZATION INDEXED, &
               MAP REC.1, &
               PRIMARY KEY SURNAME$, &
               USEROPEN OPEN_FILL
120      CLOSE #1%
130 END
```

The MACRO program OPEN_FILL is:

```
        .TITLE OPEN_FILL
        $FABDEF                         ; Define all FAB bits and offsets
        $RABDEF                         ; Define all RAB bits and offsets
        $XABDEF                         ; Define all XAB bits and offsets
        $XABKEYDEF                      ; Define all XABKEY bits and offsets
        .ENTRY OPEN_FILL, ^M<>          ; Routine to open file with bucket fill
                                        ; Percentage set to 256 bytes/bucket
;+
; Set bucket fill percentage
;-
        MOVL 4(AP),R0                   ; Get address of FAB into R0
        MOVL FAB$L_XAB(R0),R0           ; Get address of XAB into R0
10$:    CMPB #XAB$C_KEY,XAB$B_COD(R0)   ; Is this the KEY XAB?
        BEQL 20$                        ; Yes, go store fill size
        MOVL XAB$L_NXT(R0),R0           ; Get address of next XAB into R0
        BNEQ 10$                        ; If not zero, loop back for next XAB
;+
; Here if no XABKEY was found, return illegal organization
;-
        MOVL #RMS$_ORG,R0
        RET
;+
; Here when XABKEY is found, set DFL
;-
20$:    MOVW #256,XAB$W_DFL(R0)         ; Store data bucket fill size (bytes)
; +
; Set bit in RAB ROP indicating RMS should use XABKEY DFL
;-
        MOVL 8(AP),R0                   ; Get address of RAB into R0
        INSV #1, #RAB$V_LOA, #1,-       ; Set LOA bit in
        RAB$L_ROP(R0)                   ; RAB ROP
;+
; Now create and connect to the file
;-
        $CREATE  FAB = @4(AP)           ; Create file
        BLBC     R0,30$                 ; Branch if error
        $CONNECT RAB = @8(AP)           ; Connect stream to file
30$: RET                                ; Return with R0 set to status
        .END
```

In PDP–11 BASIC–PLUS–2, the USEROPEN routine is called with the FAB address as the first parameter and the RAB address as the second parameter, using the standard R5 passing sequence.

On PDP–11 systems, the USEROPEN routine may use any register; however, the stack must be in the same state at routine exit as it was at routine entrance. The RMS STS status value must be passed back to the OTS in R0.

You cannot use a USEROPEN routine to fill the RBF, UBF, BKS, or CTX fields in the RAB. These fields are filled in after the USEROPEN routine returns; thus, any values placed there by the USEROPEN routine will be overwritten.

Note

You must not set Locate mode when using a USEROPEN routine on sequential files.

The following is an example of the USEROPEN clause for PDP–11 systems. This USEROPEN routine sets the protection of the file:

```
        .TITLE  USR
;+
;       This routine will link a protection XAB to
;       the end of a linked list of XABs so that the
;       file will be created with a protection code different
;       from the default protection code for the disk it is on.
;
; INPUT:
;       2(R5) - Address to the FAB
;       4(R5) - Address to the RAB
;
; OUTPUT:
;       R0 - the STS field of either the FAB or RAB
;
; EFFECT:
;       The file is created and a connect is done if no errors occurred
;
;       R1 - R3 are destroyed.
;
; EXTERNALS:

        .MCALL  $GNCAL,FAB$B,RAB$B,XAB$B,NAM$B,$FBCAL,$RBCAL
        $GNCAL
        $FBCAL
        $RBCAL
;
;-

USR::
        MOV     2(R5),R2        ; Get fab pointer

;+
;       Walk down through the linked list of XABs ( if any ) and
;       insert the PRO XAB at the end.
;-

        $FETCH  R3,XAB,R2       ; Get the first XAB addr if any
        BEQ     2$              ; BR if none

1$:     $FETCH  R1,NXT,R3       ; Get the next XAB on the list
        BEQ     3$              ; If none left BR
        MOV     R1,R3           ; R3 = current XAB address
        BR      1$              ; Cont until done

2$:     $STORE  #PROCOD,XAB,R2  ; Store our XAB address in the FAB
        BR      4$              ; Cont

3$:     $STORE  #PROCOD,NXT,R3  ; Store our XAB address in the last XAB on list

4$:     $CREATE R2              ; Create the file

        MOV     O$STS(R2),R0    ; Get error status
        BLE     5$              ; Quit on error

        MOV     4(R5),R1        ; Get rab RAB pointer

        $CONNECT R1             ; Connect the RAB-FAB

        MOV     O$STS(R1),R0    ; Get error status
5$:     RETURN                  ; RETURN
```

(continued on next page)

```
;+
;       Set the protection code for the XAB
;-

PROCOD:
        XAB$B   XB$PRO

.IF     DF      RSX

        X$PRO   60942           ; (R,RWED,R,R)
.IFF
        X$PRO   40              ; Set protection

.ENDC
        XAB$E
        .END
```

9.12.12 DEFAULTNAME

The DEFAULTNAME clause in the OPEN statement lets you specify a default file specification for the file to be opened. BASIC uses the DEFAULTNAME clause for any part of the file specification that is not explicitly supplied. For example:

```
10 LINPUT "Next data file"; FIL$
20 OPEN FIL$ FOR INPUT AS FILE #5%, &
   ORGANIZATION SEQUENTIAL, DEFAULTNAME "DB2:.DAT"
```

The DEFAULTNAME clause supplies default values for the device, account, and file type portions of the file specification. Thus, typing ABC in response to the "Next data file?" prompt causes BASIC to try to open DB2:ABC.DAT.

Note that BASIC uses the DEFAULTNAME values only if you do not supply those parts of the file specification appearing in the DEFAULTNAME clause. For example, if you type DB3:ABC in response to the prompt, BASIC tries to open DB3:ABC.DAT. In this case DB3: overrides the device default in the DEFAULTNAME clause. Also note that any system defaults for device and account remain in effect unless DEFAULTNAME overrides them as well.

In VAX–11 BASIC and BASIC–PLUS–2 on RSX–11M/M–PLUS systems you can use the DEFAULTNAME clause with all file organizations. However, in BASIC–PLUS–2 on RSTS/E systems, you cannot use the DEFAULTNAME clause with block I/O files; that is, files with ORGANIZATION VIRTUAL.

Chapter 10
Compiler Directives

Compiler directives are instructions telling BASIC to perform certain operations as it translates a source 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 inclusion of listing information for selected parts of a program
- Start or stop the inclusion of cross-reference information for selected parts of a program
- Include BASIC code from another source file
- Include CDD record definitions in a BASIC program (VAX–11 BASIC only)
- Conditionally compile parts of a program
- Terminate compilation

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)
- Cannot begin in the first column
- Cannot appear within a quoted string

This chapter describes the use of these directives.

10.1 Controlling the Compilation Listing

Listing directives let you control the content and appearance of the compilation listing. There are eight compiler listing directives:

- %TITLE (Places a title string on the first line of the listing header)
- %SBTTL (Places a subtitle string on the second line of the listing header)
- %IDENT (Places an identification string on the second line of the listing header and within the object module)
- %PAGE (Causes BASIC to skip to top-of-form in the output listing)
- %NOLIST (Causes BASIC to stop accumulating information for the output listing)
- %LIST (Causes BASIC to resume accumulating information for the output listing)
- %NOCROSS (Causes BASIC to stop accumulating cross-reference information for the output listing)
- %CROSS (Causes BASIC to resume accumulating cross-reference information for the output listing)

Note that these listing control directives have no effect if no source program listing is being produced. Similarly, the %CROSS and %NOCROSS directives have no effect if no cross-reference listing is being produced. However, the %IDENT directive places the specified text in the object module whether or not a listing is produced. These directives are described in the following sections.

10.1.1 %TITLE and %SBTTL

The %TITLE directive lets you specify a line of text that appears on the first line of every page in the compilation listing. This text line normally contains the source program title and other information. The format of %TITLE is:

```
%TITLE text
```

where:

text Is a quoted string of up to 48 characters.

If the %TITLE directive is the first source text in a module, then the quoted string appears in the first line of every page of the compilation listing. Otherwise, the quoted string appears in the first line of every subsequent page in the compilation listing. That is, if BASIC encounters a %TITLE directive after it has begun creating a page in the output listing, the title information will not appear on that page. Rather, it appears on all of the following pages until it encounters another %TITLE directive.

The quoted string appears in character positions 33 to 81 in the first line of the listing header. %TITLE must appear on its own line. For example:

```
10      %TITLE 'File OPEN Subprogram -- Author Hugh Ristics'
        SUB FILSUB (STRING F_NAME)
```

The %SBTTL directive lets you specify a line of text that appears on the second line of every page in the compilation listing (beneath the title). If BASIC encounters a %SBTTL directive after it has begun creating a page in the output listing, the subtitle information will not appear on that page. Rather, it appears on all following pages until it encounters another %SBTTL or %TITLE directive.

Any number of %SBTTL directives can appear in a source file; thus, you can use subtitle text to identify parts of the source program. The format of %SBTTL is:

%SBTTL text

where:

text Is a quoted string of up to 48 characters.

The quoted string appears in the second line of the listing header, in character positions 33 to 81. This example shows the use of both %TITLE and %SBTTL directives:

```
10      %TITLE 'Payroll Program'
        %SBTTL 'Constant Declarations'
        !.
        !.
        !.
2000    %SBTTL 'Subroutines'
        !.
        !.
        !.
19000   %SBTTL 'Error Handler'
        !.
        !.
        !.
```

The first line of the listing's first page contains "Payroll Program" and the second line contains "Constant Declarations". When BASIC encounters the %SBTTL directive on line 2000, the second line on each subsequent page becomes "Subroutines". When BASIC encounters the %SBTTL directive on line 19000, the second line on each subsequent page becomes "Error Handler".

Note

You can use multiple %TITLE directives in a single source file; however, whenever BASIC encounters a %TITLE directive, the %SBTTL information is set to the null string. Therefore, if you want to display subtitle information, each new %TITLE directive should be accompanied by a new %SBTTL directive.

10.1.2 %IDENT

The %IDENT directive identifies the version of a program module. The %IDENT information appears in the listing file and the object module. Thus, the map file created by the Task Builder (on PDP–11 systems) or the VAX–11 Linker also contains this information. The format of %IDENT is:

%IDENT text

where:

text Is a quoted string of up to 6 characters in PDP–11 BASIC–PLUS–2 and up to 31 characters in VAX–11 BASIC.

The identification text appears in the first 6 to 31 character positions of the second line on each subsequent listing page. For example:

```
100     %IDENT 'V5.3'
        SUB PAY
        !.
        !.
        !.
```

The %IDENT information appears as the first entry on the second line of the listing. The information is also included in the object module if the compilation produces one. If the Task Builder or Linker generates a map listing, this information also appears there.

On VAX/VMS systems, the information supplied by %IDENT can control certain operations of the VAX–11 Linker. See the *VAX–11 Linker Reference Manual* and the *BASIC on VAX/VMS Systems* manual for more information.

If your source module contains multiple %IDENT directives, BASIC signals a warning and uses the version specified in the first %IDENT directive.

10.1.3 %PAGE

The %PAGE directive causes BASIC to begin a new page in the listing file. Its format is:

```
%PAGE
```

For example:

```
10      %TITLE 'Payroll Program'
        %PAGE
        %SBTTL 'Constant Declarations'
        !.
        !.
        !.
2000    %PAGE
        %SBTTL 'Subroutines'
        !.
        !.
        !.
19000   %PAGE
        %SBTTL 'Error Handler'
        !.
        !.
        !.
```

The %PAGE directives cause BASIC to skip to a new page in the listing file just before each new subtitle. Note that, in order to have title and subtitle information appear in the heading of this page, you cannot place a line number between the %PAGE, %TITLE, and %SBTTL directives.

10.1.4 %LIST and %NOLIST

%NOLIST causes BASIC to stop adding information to the output listing file while %LIST causes BASIC to resume adding information to the listing file. %LIST and %NOLIST are complementary directives. Thus, you can control which parts of the source program are to be listed. The format of %LIST is:

```
%LIST
```

The format of %NOLIST is:

```
%NOLIST
```

For example:

```
10      %TITLE 'Payroll Program'
        %PAGE
        %SBTTL 'Constant Declarations'
        !.
        !.
        !.
100     %NOLIST
        !.
        !.
        !.
1900    %LIST
        !.
        !.
        !.
2000    %PAGE
        %SBTTL 'Subroutines'
        !.
        !.
        !.
19000   %PAGE
        %SBTTL 'Error Handler'
        !.
        !.
        !.
```

The %NOLIST directive in line 100 causes BASIC to stop adding new information to the listing file. The %LIST directive in line 1900 causes BASIC to resume adding this information, and the directive itself appears as the next line in the listing file. Thus, BASIC produces no listing information for lines 100 to 1900.

Note

If you have not requested the creation of a compilation listing, the %LIST and %NOLIST directives have no effect.

If a program line contains a syntax error, BASIC overrides the %NOLIST directive for that line and produces the normal error diagnostics in the listing file.

10.1.5 %CROSS and %NOCROSS

%NOCROSS causes BASIC to stop adding cross-reference information to the output listing file while %CROSS causes BASIC to resume adding cross-reference information. Thus you can specify that only certain parts of the source program are to be cross-referenced. The format of %CROSS is:

```
%CROSS
```

The format of %NOCROSS is:

```
%NOCROSS
```

For example:

```
10      %TITLE 'Payroll Program'
        %PAGE
        %SBTTL 'Constant Declarations'
        !.
        !.
        !.
```

(continued on next page)

```
100     %NOCROSS
        !.
        !.
        !.
1900    %CROSS
        !.
        !.
        !.
2000    %PAGE
        %SBTTL 'Subroutines'
        !.
        !.
        !.
19000   %PAGE
        %SBTTL 'Error Handler'
        !.
        !.
        !.
```

The %NOCROSS directive in line 100 causes BASIC to stop adding new cross-reference information to the listing file. The %CROSS directive in line 1900 causes BASIC to resume adding this information. Thus, BASIC produces no listing information for lines 100 to 1900, inclusive.

Note

If you have not requested the creation of a cross-reference listing, the %CROSS and %NOCROSS directives have no effect.

10.2 Accessing External Source Files (%INCLUDE)

The %INCLUDE directive lets you copy BASIC source text from a file into the source program. On VAX/VMS systems, %INCLUDE also lets you access record definitions in the VAX–11 Common Data Dictionary (CDD). The line on which a %INCLUDE resides can be continued but cannot contain any other directives or statements. The format of %INCLUDE is:

```
%INCLUDE  { file-spec                          }
          { %FROM %CDD cdd-path-spec           }
```

where:

file-spec Is any valid file specification. BASIC uses the default device and directory if you do not supply these parts of the file specification. If you do not supply a file type, the default is BAS in VAX–11 BASIC and B2S in BASIC–PLUS–2.

cdd-path-spec Is valid only in VAX–11 BASIC and specifies a VAX–11 CDD path specification. This lets you extract a RECORD definition from the CDD. See the *BASIC on VAX/VMS Systems* manual and the *VAX–11 Common Data Dictionary Utilities Reference Manual* for more information.

File specifications and CDD path specifications must be string literals enclosed in quotes.

The source files accessed with %INCLUDE cannot contain line numbers. Also, each line in the file must begin with a space or a tab. These requirements mean that all statements in the accessed file are associated with the BASIC line containing the %INCLUDE directive. Thus, a %INCLUDE directive cannot appear before the first line number in a source program. A file accessed by %INCLUDE can itself contain a %INCLUDE directive.

%INCLUDE is useful when you want to share code among multiple program modules. When you insert code into a file and then include the file in all the modules that require it, you reduce the chance of a typographical error. For example, this code might be contained in a file named SHRVAL.BAS:

```
COMMON (SHARED) LONG FLAG_ARRAY(100),  &
                BYTE BYTE_ARRAY(10),   &
                STRING ERR_MSG,        &
                       SUCCESS_MSG
```

Each program module that requires this common area can access it with the %INCLUDE directive:

Main Program

```
1000    %INCLUDE 'SHRVAL.BAS'
```

Subprogram

```
2000    %INCLUDE 'SHRVAL.BAS'
```

When these programs are compiled, BASIC places the text from SHRVAL.BAS into each source program at the point of the %INCLUDE directive. The compilation listing identifies any text obtained from an included file by placing a mnemonic in the first character position of the line in which the text appears. The mnemonic is of the form:

In

where:

I Tells you that the text was accessed with a %INCLUDE directive.

n Tells you the nesting level of the included text.

You can prevent the %INCLUDE file code from appearing in the compilation listing by preceding the %INCLUDE directive with a %NOLIST directive.

10.3 Controlling Compilation

BASIC lets you control the compilation of a program by creating and testing *lexical constants*. You create and assign values to lexical constants with the %LET directive. These constants are always integers, LONG integers in VAX–11 BASIC and WORD integers in PDP–11 BASIC–PLUS–2.

You control the compilation by using the %IF–%THEN–%ELSE–%END–%IF directive to test these lexical constants. Thus, you can conditionally:

- Supply different values for program variables and constants
- Skip over part of a program
- Abort a compilation
- Include BASIC source code from another file

BASIC also supplies the lexical built-in function %VARIANT, which can be used to conditionally control compilation. See Section 10.3.2.

%IF–%THEN–%ELSE–%END–%IF uses *lexical expressions* to determine whether to execute directives in the %THEN clause or the %ELSE clause. The following sections describe the use of:

- Lexical constants and expressions (%LET)
- %ABORT
- %VARIANT
- %IF–%THEN–%ELSE–%END–%IF

10.3.1 Lexical Constants and Expressions (%LET)

The %LET directive creates and assigns values to lexical constants. Lexical constants are always integers, LONG integers in VAX–11 BASIC, and WORD integers in PDP–11 BASIC–PLUS–2. These constants control the execution of the %IF–%THEN–%ELSE–%END–%IF directive. Lexical constants must be created with %LET before they can be used in a %IF–%THEN–%ELSE–%END–%IF, and each lexical constant must be created with a separate %LET directive. The format of %LET is:

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

where:

%lex-const-nam	Is the name of a lexical constant. Lexical constant names must be preceded by a percent sign and cannot end with a dollar sign or percent sign. Each lexical constant must be declared with its own %LET directive, and a %LET directive can name only one lexical constant.
lex-exp	Is a lexical expression.

A lexical expression can be:

- A lexical constant
- An integer literal
- A lexical built-in function (%VARIANT)
- Any combination of these, separated by logical, relational, or arithmetic operators

The %LET directive lets you create constants that control conditional compilation. For example:

```
100     %LET %DEBUG_ON = 0%
```

See Section 10.3.4 for an example of using %LET with %IF–%THEN–%ELSE.

10.3.2 %VARIANT

%VARIANT is a built-in lexical function that returns an integer. The value of this returned integer is determined by:

- The SET VARIANT command when compiling in the BASIC environment
- The /VARIANT qualifier when compiling from the system command level (VAX–11 BASIC only)

The %VARIANT function returns the variant value set with either of these methods. Its format is:

```
%VARIANT
```

The default value for the %VARIANT function is zero. See Section 10.3.4 for an example of controlling compilation with %VARIANT.

10.3.3 %ABORT

The %ABORT directive terminates the compilation and displays a message you provide. Its format is:

```
%ABORT text
```

where:

text Is a quoted string literal.

The text is displayed on the terminal screen and in the compilation listing if one is being created. Note that BASIC stops the compilation and terminates the listing file as soon as it encounters a %ABORT directive. This means that BASIC does not perform syntax checking on the remainder of the program. See the following section for an example of using %ABORT.

10.3.4 %IF–%THEN–%ELSE–%END–%IF

The %IF–%THEN–%ELSE–%END–%IF directive lets you conditionally: 1) place text in a source program or 2) execute another compiler directive. Its format is:

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

where:

lex-exp Is a lexical expression.

source-code Is BASIC source text. Note that compiler directives are also BASIC source code.

This directive differs from all others in that it can appear anywhere in a program where a space is allowed, except in column one or within a quoted string.

You must include %END %IF. Otherwise, the rest of the source program becomes part of the %THEN or %ELSE clause.

The truth or falsity of the lexical expression determines whether BASIC compiles the source code in the %THEN clause or the %ELSE clause. If the lexical expression is true, BASIC neither compiles nor checks the syntax of source code in the %ELSE clause. If the lexical expression is false, BASIC neither compiles nor checks the syntax of source code in the %THEN clause.

The following example also uses the %VARIANT directive, which returns the value set by the SET VARIANT command or /VARIANT qualifier.

```
100     %IF (%VARIANT = 2%) %THEN DECLARE LONG INT_ARRAY(100)
                            %ELSE DECLARE WORD INT_ARRAY(100)
        %END %IF
```

This directive allows for two possibilities: If you compile this program with a /VARIANT=2 qualifier, then BASIC creates an array of longword integers. If you compile this program with any other variant value, BASIC creates an array of word integers.

Because %IF can appear within a program line, you can express the same directive this way:

```
10 DECLARE %IF (%VARIANT=2%) %THEN LONG %ELSE WORD %END %IF INT_ARRAY(10)
```

A %THEN or %ELSE clause can also contain another compiler directive. For example:

```
100     %LET %my_constant = 8%
        %IF ( (%my_constant + %VARIANT) >= 10% )
            %THEN %ABORT 'Cannot compile with VARIANT >= 2'
        %END %IF
```

This example creates the lexical constant %my_constant and assigns it a value of eight. The %IF directive evaluates the conditional expression (%my_constant + %VARIANT) => 10%. If this expression is true, BASIC executes the %THEN clause, aborting the compilation and issuing an error message. If false, the compilation continues.

The compilation listing shows you which clause was actually compiled.

Chapter 11
Handling Run-Time Errors

This chapter describes how to handle run-time errors.

11.1 Errors

If an error occurs when your program is running (at "run-time"), BASIC provides two ways to help you correct or circumvent the condition causing the error.

- BASIC error handling: BASIC diagnoses the error and displays a message telling the nature and severity of the error, and the program line and module that caused it. This information lets you correct whatever caused the error before you try running the program again.
- User error handling: For most errors, BASIC lets you handle the error in your own routine. This error routine can test the error condition and take appropriate action.

There are three general error categories: 1) informational or warning errors, 2) trappable errors, and 3) fatal errors.

The severity of an error determines whether or not the program aborts when the error occurs. Informational and warning errors let the program keep running; fatal errors always terminate program execution. Trappable errors can have either result, depending on the error handler in effect: the BASIC error handler terminates program execution, but a user-written error handler may let your program keep running.

11.2 Error Handlers

An error handler is a block of code that receives program control in the event of an error or unexpected event. If you do not supply an error handler, the BASIC error handler usually gets program control when an error occurs (there are some types of errors that are handled only by the operating system).

Depending on the severity of the error, the BASIC error handler either: 1) reexecutes the statement in which the error occurred, 2) issues a warning and continues execution, or 3) terminates the program.

In a BASIC program, there are many possible levels of error handling. These software components and program modules can each have an error handler:

- Mainline code
- User-written subprograms or external functions
- User-defined functions (DEFs)
- The BASIC system

You can supply your own error handler for each of the first three components. The BASIC error handler is the default if you do not supply one. When an error occurs, BASIC stops executing the program and transfers control to the BASIC error handler. If your program includes a user-written error handler, and the error is trappable, the BASIC error handler transfers control to the user error handler. If your program does not include a user error handler, or the error is a nontrappable or fatal error, the BASIC error handler retains control.

Any program module can contain one or more user-written error handlers. A user-written error handler normally performs these functions:

- Determines the BASIC error number associated with the error
- Determines the program module in which the error was detected
- Determines the program line in which the error was detected
- Takes appropriate action based on the above information
- Clears the error condition and resumes program execution

In BASIC, only one error can be handled at a time. If an error is pending and a second error occurs, program execution always terminates immediately. Therefore, one of the most important functions of a user-written error handler is to clear the error condition so that subsequent errors can also be handled.

11.3 User-Written Error Handlers

The only way to prevent a trappable error from aborting your program is to provide a user error handler. When an error occurs during the execution of a program for which you have written an error handler, BASIC branches to your error-handling routine, which processes the error. After processing the error, the error handler can:

- Resume execution at a specified line (RESUME with line number)
- Resume execution at the point where the error was detected (RESUME with no line number)
- Pass the error to the BASIC error handler (ON ERROR GOTO 0)
- Pass the error to the calling program (ON ERROR GO BACK)

You can provide error handlers in your main program, subprogram (SUB or FUNCTION), or user-written function (DEF).

11.3.1 Transferring Control to an Error Handler (ON ERROR Statement)

If no error has occurred, the ON ERROR statement specifies the branch to be taken in the event of an error. If an error has occurred, the ON ERROR statement immediately transfers control to the specified error handler. ON ERROR has the following formats:

ON ERROR GOTO { 0 / line-number / label }

or

ON ERROR GO BACK

Note that, if an error has occurred and has not been handled, the ON ERROR statement immediately transfers control to the specified line number or label, or to the BASIC error handler.

ON ERROR GOTO 0 specifies that the BASIC error handler receives control when an error occurs. ON ERROR GOTO line number or label specifies that the block of code at the target line number or label is to get control when an error occurs. ON ERROR GO BACK is normally executed in subprograms or DEFs and specifies that the error handler in the calling program receives control when an error occurs. If you use ON ERROR GO BACK in a main program (outside of a DEF), it is equivalent to ON ERROR GOTO 0.

Customarily, the ON ERROR GOTO statement occurs before any other executable statements, and the error handling routines usually start at line 19000. For example:

```
5 ON ERROR GOTO 19000
```

When your program generates an error, control transfers to the specified line. If your error handler routine generates a second error, control returns to BASIC error handling and program execution ends, usually with the first error only partly processed. To avoid the possibility of your error handler causing a second error, make it as simple as possible.

The ON ERROR GOTO statement remains in effect after your program successfully handles an error. When the system signals another error, control once again transfers to the specified line.

Your routine can use conditional expressions to test the error and branch accordingly. For example:

```
5       ON ERROR GOTO 19000
                !.
                !.
                !.
19000   SELECT ERR
        CASE 161
                PRINT "Record too long"
                RESUME 650
        CASE 11
                PRINT "End of file"
                RESUME 32000
        CASE ELSE
                ON ERROR GOTO 0
        END SELECT

32000   CLOSE #1%
32767   END
```

This routine tests for two errors: 1) records longer than the buffer length and 2) end of file.

For some conditions that generate errors, for example, "End of File", you can keep the program from aborting by writing a user error handler that traps the error. That is, the routine provides BASIC statements that deal with the error and then instructs BASIC to resume execution at a certain program line. For "End of File" you can display a message indicating that the task is finished and then transfer control to the END statement.

It is good programming practice to anticipate certain errors and provide error handlers for them. Your error handler can first test whether or not the error is the one you expect and then execute the appropriate BASIC statements.

Two common errors you can trap with error handlers are "Division by 0" and "Data format error". If your program is reading data from a file when one of these errors occurs, you can have it print an error message and skip to the next item; if the program is reading data the user types in, you can display a "Try again" message and reexecute the program lines requesting input.

Normally, you cannot trap fatal errors in an error handler, nor can you trap errors occurring in non–BASIC modules. However, in BASIC–PLUS–2 you can trap certain errors (for example, CTRL/C interrupts) occurring in a MACRO–11 subprogram. See *BASIC on RSTS/E Systems* and *BASIC on RSX–11M/M–PLUS Systems* for more information about MACRO–11 subprograms.

11.3.2 Determining the Error Number (ERR)

The ERR function returns the number of the last error that occurred. Accessing the ERR function is a valid operation only inside an error handler. For example:

```
19000   PRINT "ERROR NUMBER ";ERR
19010   SELECT ERR
        CASE 153
                PRINT "Choose new record"
                RESUME 420
        CASE ELSE
                ON ERROR GO BACK
        END SELECT
```

This routine transfers control to line 420 when the program generates error 153, "Record already exists". Line 19000 prints the value of ERR each time BASIC traps an error. See Appendix B for the number of each BASIC run-time error.

The results of ERR remain undefined until an error occurs. Although ERR remains defined as the number of the last error after control leaves the error handler, it is poor programming practice to refer to this variable outside the scope of an error handler because it can be changed at any time by an asynchronous error.

11.3.3 Determining the Error Line Number (ERL)

After your program generates an error, the ERL function returns the line number of the signaled error. The ERL function, like ERR, lets you set up branching to one of several routines. For example:

```
1       ON ERROR GOTO 19000
        DECLARE INTEGER CONSTANT TRUE = -1
        !.
        !.
        !.
```

```
19000   SELECT TRUE
        CASE (ERR = 11) AND (ERL = 790)
        !Is error end-of file at line 790?
                PRINT "Completed"
                RESUME 32000
        CASE (ERR = 149) AND (ERL = 80)
        !Is error not-at-end-of-file on line 80?
                PRINT "CHECK ACCESS MODE"
                RESUME 32000
        CASE ELSE
        !Let BASIC handle any other errors
                ON ERROR GOTO 0
        END SELECT
32000   CLOSE #5
32767   END
```

This routine performs three tests:

1. If the error is "End of file" (ERR = 11), it closes the file and exits.
2. If the error is "Not at end of file" (ERR = 149), it returns control to line 50.
3. If the error is anything else, it passes control to the BASIC error handler.

The results of ERL are undefined: 1) until an error occurs and 2) if the error occurred in a subprogram not written in BASIC. Although ERL remains defined as the line number of the last error after control leaves the error handler, it is poor programming practice to refer to this variable outside the scope of an error handler.

Also, if your program references ERL and you compile the program with the /NOLINE switch, BASIC signals "ERL overrides /NOLINE," and the compilation continues.

If an error occurs in a subprogram, BASIC sets the ERL variable to the subprogram line number where the error was detected. Thus, if the subprogram executes an ON ERROR GO BACK statement, the error handler in the calling program cannot execute a RESUME without specifying a line number because you cannot resume execution at a line outside the current program module. Instead, the program should resume execution at the line containing the CALL to the subprogram. For example:

Main Program

```
1 ON ERROR GOTO 19000
        !.
        !.
        !.
5000    CALL SUB1
        !.
        !.
        !.
19000   IF (ERN$ = SUB1) AND (ERR = 138) !Is file locked?
        THEN
            SLEEP 5
            RESUME 5000
        ELSE
            ON ERROR GOTO 0
        END IF
```

This error handler checks for a locked file condition. If the error number is 138, the error handler waits 5 seconds and then resumes execution. The RESUME transfers control to the line 5000, which calls SUB1 again.

11.3.4 Determining Where the Error Occurred (ERN$)

The ERN$ function returns the name of the program unit in which the error was detected. In BASIC–PLUS–2, ERN$ returns the name of a main program or subprogram, whereas, in VAX–11 BASIC, ERN$ returns the name of a main program, subprogram, or DEF function. The results of ERN$ are undefined until the program generates an error. For example:

```
19000    IF ERN$ = "SUBARC" THEN PRINT "ERROR IS ";ERR
                 PRINT "RETURNING TO MAIN PROGRAM FOR ERROR HANDLING"
                 ON ERROR GO BACK
19010    PRINT "PROGRAM MODULE GENERATING ERROR IS ";ERN$
19020    IF ERR = 153% THEN RESUME 32000
                 ELSE ON ERROR GO TO 0
32000 CLOSE #2%
32767 END
```

In this example:

1. Control passes to the main program for error handling if the error occurs in the module SUBARC.

2. Execution resumes at line 32000 if the error is number 153, "?Record already exists".

3. Control passes to BASIC error handling for all other errors.

11.3.5 Determining the Error Message Text (ERT$)

The ERT$ function returns the message text associated with a specified error number. For example:

```
19000    IF ERN$ = "TSLFE"
                 THEN PRINT ERT$(ERR)
                 RESUME
```

This statement tests whether the error occured in a DEF module named TSLFE, and if so, prints the text of the signaled error and resumes execution. Note that if an error occurs in a DEF, a RESUME with no line number causes execution to resume at the statement that invoked the function. Use of the ERT$ function is not limited to the scope of the error handler; you can access ERT$ at any time.

11.4 Returning to BASIC Error Handling

The ON ERROR GOTO 0 statement disables program error trapping and returns control to BASIC error handling. The BASIC error handler displays error messages, stops program execution, and prints a fatal error message.

ON ERROR GOTO 0 returns control to BASIC error handling in one of two ways:

- If an error is pending, execution of the ON ERROR GOTO 0 statement returns control to BASIC error handling immediately.

- If no error is pending, an ON ERROR GOTO 0 statement disables your error handler. The BASIC error handler handles all subsequent errors until another ON ERROR statement is executed.

For example:

```
10    ON ERROR GOTO 19000
20    OPEN 'FILE.LIS' FOR INPUT AS FILE #2%
30    LINPUT #2%, A$
40    PRINT A$
50    GOTO 30
19000 IF (ERR = 11%) AND (ERL = 30%)
      THEN
       RESUME 32000
      ELSE
       ON ERROR GOTO 0
      END IF
32000 CLOSE #2%
32767 END
```

In this program, line 10 establishes the error routine. Lines 30, 40, and 50 retrieve and print lines. When an error occurs, BASIC transfers control to line 19000. If the error is end-of-file, the error handler resumes to line 32000, which closes the file. If any other error occurs, the ELSE clause transfers control to the BASIC error handler, which prints information about the error.

You can use a mix of ON ERROR GOTO and ON ERROR GOTO 0 statements to turn your program's error handler on and off. In this way, your program can handle certain errors, and BASIC can handle the rest.

11.5 Leaving an Error Handler

The RESUME statement clears the error condition and returns control to the specified line number or to the program block in which the error occurred. An error handler must end with a RESUME or an ON ERROR GOTO 0 or ON ERROR GO BACK statement. If it does not, the BASIC error handler aborts your program with the fatal error "Error trap needs RESUME" as soon as an END, END SUB, END DEF, or END FUNCTION is encountered. In this example, all paths out of the error handler are through either an ON ERROR or RESUME statement:

```
19000       IF (ERR = 11%) AND (ERL = 30%)
            THEN
                    CLOSE #2%
                    RESUME 32767
            ELSE
                    ON ERROR GOTO 0
            END IF
32000       CLOSE #2%
32767       END
```

You can resume execution at any line number that is in the same module as the RESUME statement, unless that line is inside a DEF. Note that the RESUME statement does not accept a label as an argument. Therefore, you should number lines that receive control from the error handler.

In general, RESUME without a line number transfers control to the beginning of the program block where the error occurred. A program block can begin with either a line number or a label. Thus, if you resume execution at a multi-statement line, execution begins at the the first statement after the line number or label — not necessarily at the statement that generated the error.

However, if an entire loop block is associated with a single line number or label and an error occurs within the loop, RESUME with no line number transfers control to the statement immediately after the FOR, WHILE, or UNTIL statement, not to the line number or label. For example:

```
1         ON ERROR GOTO 19000
100       FOR I% = 1% TO 10%
              PRINT "Please type a number"
              INPUT A
          NEXT I%
19000     IF (ERR = 50)
          THEN
              RESUME
          ELSE
              ON ERROR GOTO 0
          END IF
```

In this example, if 'ABC' is typed in response to the INPUT prompt, control is transferred to line 19000, which traps error number 50 and executes a RESUME statement with no line number. The RESUME statement transfers control to the first statement following the FOR statement. In the case of a FOR loop, the loop control variables (the starting, ending, and STEP values) are not reinitialized when the loop resumes execution. Similarly, if an entire WHILE or UNTIL loop is associated with a single line number, RESUME with no line number transfers control to the first statement following a WHILE or UNTIL statement.

If a SELECT block is associated with a single line number or label and an error occurs within the SELECT block, RESUME with no line number transfers control to the start of the CASE block in which the error occurred. For example:

```
1         ON ERROR GOTO 19000
10        INPUT "Which parameter do you want to change"; PARM%
100       SELECT PARM%
          CASE = 1
              PRINT "What is the new parameter value"
              INPUT PARM_1
          CASE = 2
              PRINT "What is the new parameter value"
              INPUT PARM_2
          END SELECT
200       GOTO 32767
19000     IF (ERR = 50)
          THEN
              RESUME
          ELSE
              ON ERROR GOTO 0
          END IF
32767     END
```

In this example, if 'ABC' is typed in response to one of the INPUT prompts within the SELECT block, control is transferred to line 19000. The error handler traps error number 50 and executes a RESUME statement with no line number. The RESUME statement transfers control to the first statement in the CASE block in which the error occurred.

Finally, if an error occurs after a loop or SELECT block in a multi-statement line, RESUME without a line number transfers control to the next statement after the NEXT or END SELECT statement.

If you use RESUME without a line number, you must take care to supply a separate line number for every statement that may generate an error. This is especially important for loop and SELECT blocks. If you number every statement in the loop or SELECT block, you can be sure RESUME with no line number transfers control to the statement that caused the error.

If you use /NOLINE to compile a program containing a RESUME without a line number, you receive the warning message, "RESUME overrides /NOLINE".

If you use GOTO from an error handler, you leave the error condition pending and the next error aborts the program. To prevent this, use a RESUME, an ON ERROR GO BACK, or an ON ERROR GOTO 0 statement instead of GOTO.

Similarly, even though the line at which you want to resume program execution directly follows the error handler in the program, you cannot "fall through" the error handler to a END SUB, END DEF, END FUNCTION, or END statement. For example, these error handlers are invalid:

```
19000    IF (ERR = 11%) AND (ERL = 550%)
         THEN
                 PRINT "WRITING COMPLETE"
                 CLOSE #5%
                 PRINT "FILE CLOSED"
                 PRINT "RETURNING TO MAIN PROGRAM"
         END IF
32767    END SUB

19000    IF (ERR = 132%)
         THEN
                 A% = A% + 1%
                 GET #2%, RECORD A%
                 PRINT NA.ME.CODE$, ACTION.CATE%
         ELSE
                 PRINT "NEXT RECORD INVALID"
         END IF
32767    END
```

These error handlers are invalid because they pass control to the END SUB or END statements without handling the error (that is, without resuming to a line number or returning to system error handling). In both cases, the error condition is still pending as the module ends.

You must also make sure that your error handler does not return control without correcting or removing the condition that caused the error. Otherwise, the program generates the error and handles it unsuccessfully over and over. For example:

```
5       ON ERROR GOTO 19000
                !.
                !.
                !.
340     FIND #12%, KEY #3% EQ TARGET.NAME$
                !.
                !.
                !.
19000   IF (ERR = 153%)
        THEN
                PRINT "NAME INVALID"
        END IF
19010   RESUME 340
                !.
                !.
                !.
32767   END
```

This program searches again and again for the record with a field equal to the variable TARGET.NAME$. Each time, the error handler returns control to the statement that generated the error. The error handler must: 1) handle the error in the routine or 2) return control to another program line. For example:

```
5       ON ERROR GOTO 19000
                !.
                !.
                !.
330     INPUT "WHAT RECORD DO YOU WANT"; TARGET.NAME$
340     FIND #12%, KEY #3% EQ TARGET.NAME$
                !.
                !.
                !.
19000   IF (ERR = 153%) THEN PRINT "NAME INVALID"
19010   RESUME 330
                !.
                !.
                !.
32767   END
```

This program searches for a target name, notes when the name is not in the file, and returns control to line 330 to prompt for a new name.

Note

In VAX–11 BASIC, a RESUME statement that branches to the end of the program always returns a STATUS of success to command level — even if your handler did not trap the error successfully. In some cases, (for example, controlling events in a batch stream), this is not desirable. If the error handler must return the true error handling STATUS, end the handler with ON ERROR GOTO 0 or ON ERROR GO BACK.

11.6 Handling Errors in Subprograms and Function Definitions

You can write error handlers for subprograms and function definitions (both FUNCTION subprograms and DEF function definitions) just as you do for main programs. In these error handlers, the ON ERROR GOTO and ON ERROR GOTO 0 statements behave just as they do in main programs. In addition, the ON ERROR GO BACK statement lets you transfer control to the calling program's error handler if an error occurs in the subprogram or DEF. For example:

```
10      SUB LIST(A$)
20      ON ERROR GO TO 19000
30      OPEN A$ FOR INPUT AS FILE #12%
40      LINPUT #12%, B$
50      PRINT B$
60      GO TO 40
19000   IF (ERR = 11%) AND (ERL = 40%) !END OF FILE
        THEN
                RESUME 32000
        ELSE
                ON ERROR GO BACK
        END IF
32000   CLOSE #12%
32767   END SUB
```

Note

Although you can CALL a subprogram while an error is pending, if you do, the subprogram cannot execute an ON ERROR GO BACK or an ON ERROR GOTO 0 statement. If the subprogram tries to execute one of these statements, BASIC signals "Improper error handling" and program execution terminates.

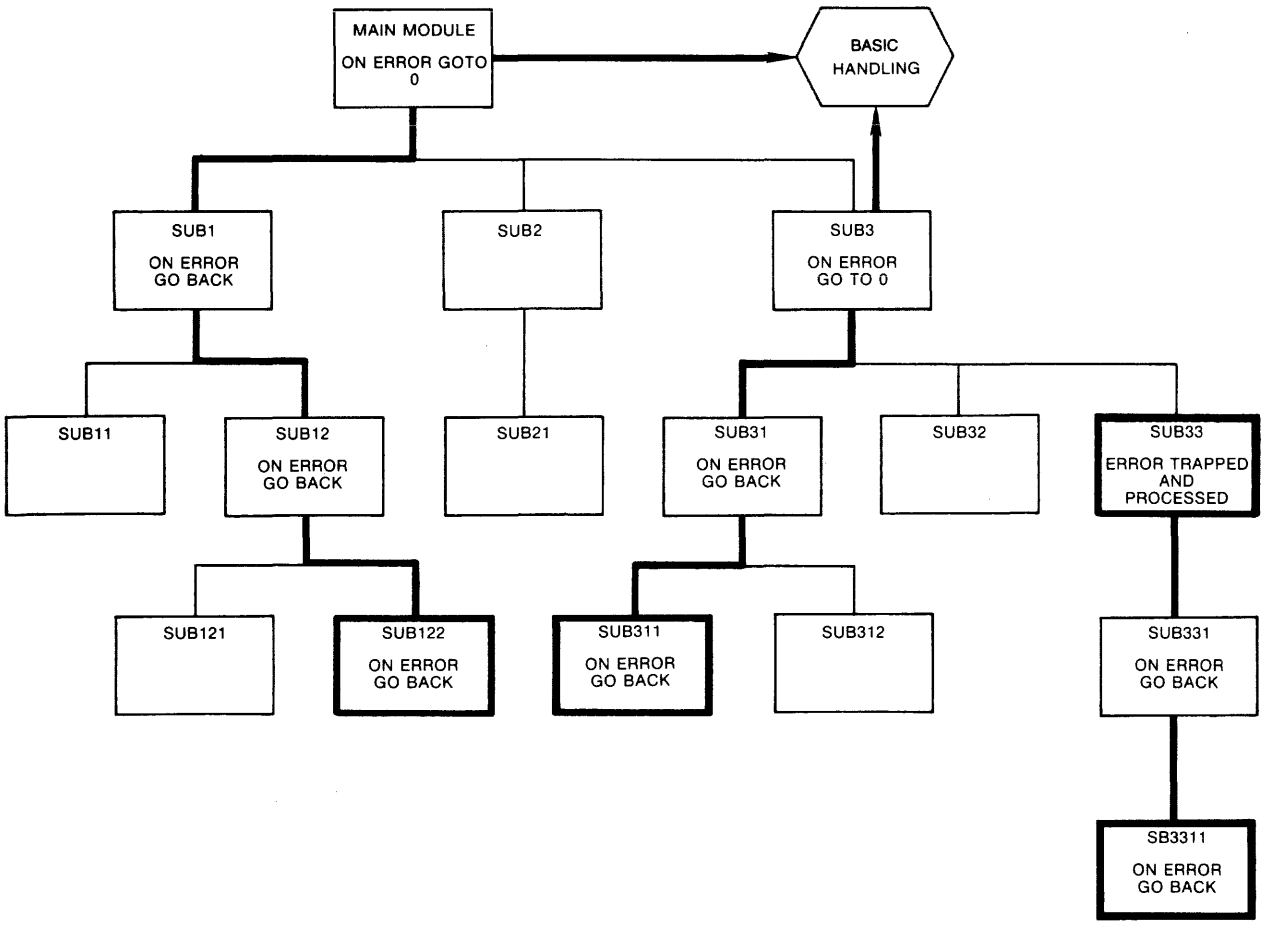
The default error handler for functions is ON ERROR GO BACK. This means that, if an error occurs in the execution of the function, the error is passed to either: 1) the error handler of the surrounding program module (in the case of a DEF function definition) or 2) the error handler of the calling program (in the case of a separately compiled subprogram). In either case, you can use the ON ERROR GOTO statement to set up an explicit error handler for the function. The subprogram can have an error handler similar to:

```
19000   IF ERR = 11
        THEN
                RESUME 1000
        ELSE
                ON ERROR GO BACK
        END IF
```

In this subprogram, the user error routine handles end-of-file conditions. Control passes to the main program for all other errors.

When your program executes an ON ERROR GO BACK, BASIC returns to the BASIC error handler or to another user error handler depending on the location of the ON ERROR GO BACK. Figure 11–1 shows three sets of error handling conditions.

Figure 11–1: Error Handling Conditions



The first error occurs in module SUB122. That module has an ON ERROR GO BACK in effect, so control returns to the calling module, SUB12. This module also has ON ERROR GO BACK in effect, so control returns to its calling module, SUB1. SUB1 also has an ON ERROR GO BACK that returns control to the main module, which contains an ON ERROR GOTO 0. At this point, the BASIC error handler operates and, if the error is fatal, stops the program. The error message identifies SUB122 as the error location. The traceback shows that control was in the main module when it found an ON ERROR GOTO 0.

The second error occurs in module SUB311. That module has an ON ERROR GO BACK in effect, so control returns to the calling module, SUB31. SUB31 also has an ON ERROR GO BACK, so control returns to its calling module, SUB3. SUB3 contains an ON ERROR GOTO 0, and the BASIC error handler comes into play. The error message identifies SUB311 as the location of the error, and the traceback shows the main module called SUB3. BASIC executed no statements in the main module.

The third error occurs in SB3311. SB3311 contains an ON ERROR GO BACK. BASIC returns to SUB331, which also contains an ON ERROR GO BACK. This returns control to SUB33. SUB33 contains an error trap for this error and successfully processes it. The program can continue execution. The execution continues in SUB33, where the error was processed, and not in SUB331 or SB3311.

For error handling in function definitions, these rules apply:

1. An error handler outside the definition cannot return control to a line inside the DEF.

2. To trap errors while the DEF is active, include error handlers inside the DEF. They remain in effect until your program leaves the DEF.

3. An error handler in the DEF does not permanently override an error handler in the main program. BASIC saves the error handler in the main program when you transfer into a DEF, and restores it when you return.

For example:

```
10      ON ERROR GOTO 19000
                !.
                !.
                !.
1000    A% = FNIN_PUT%("PROMPT")
                !.
                !.
                !.
15000   DEF FNIN_PUT%(P%)
                ON ERROR GOTO 15090
15010           PRINT P$
                INPUT LINE_IN$
15020   FNIN_PUT% = VAL%(LINE_IN$)
                FNEXIT
15090   IF ERL = 15010
        THEN
                PRINT "RETRY"
                RESUME 15010
        ELSE
                ON ERROR GO BACK
                !.
                !.
                !.
16000   END DEF
19000   PRINT "ERROR"; ERT$(ERR);
                IF ERN$ = "FNIN_PUT" THEN
                PRINT "IN FUNCTION"
                RESUME 1200
                ELSE PRINT "IN MAIN"
                RESUME 1000
32767   END
```

11.7 CTRL/C Trapping

With control C trapping enabled, control is transferred to an error handler when you type ^C (CTRL/C) during program execution. You enable CTRL/C trapping in your program by invoking the CTRLC function. For example:

```
5       ON ERROR GOTO 19000
10      Y% = CTRLC
```

After invoking CTRLC, all CTRL/C errors transfer control to the error handler.

After trapping the error, you can then include routines to interact with the program. For example:

```
5       ON ERROR GOTO 19000
10      Y% = CTRLC
                !.
                !.
                !.
460     OPEN 'FIL.DAT' FOR INPUT AS FILE #1%
470     INPUT "HOW MANY RECORDS"; REC.READ%
480     FOR I% = 1% TO REC.READ%
490             GET #1%
500             PRINT NA.ME$, ADDRE.SS$, EMP.CODE%
                PRINT
510     NEXT I%
                !.
                !.
                !.
19000   IF (ERR = 28%)
                THEN
                Y% = CTRLC
                PRINT "CURRENT RECORD IS "; I%
        ELSE ON ERROR GOTO 0
19010   INPUT "DO YOU WISH TO END PROCESSING"; ANSWER$
19020   IF ANSWER$ = "YES"
                THEN RESUME 32000
                ELSE RESUME
                !.
                !.
                !.
32000   CLOSE #1%
32766   PRINT "END OF PROCESSING"
32767   END

RUN

SMITH, DEXTER 231 COLUMBUS ST                   09341

TRAVIS, JOHN  PO BOX 80                         64119

^C

THE CURRENT RECORD IS 3

DO YOU WISH TO END PROCESSING? YES
```

Note that the error condition is still pending until the error handler executes the RESUME statement. Therefore, if you type a second CTRL/C while the error handler is executing, control returns to the BASIC error handler, which terminates the program.

RESUME with no line number returns control to the line where the error occurred. Your program can then re-execute statements interrupted by the CTRL/C. In contrast, RESUME with a specified line number can generate unpredictable results. For example, in this program, a CTRL/C at line 550 can leave the value of STRINGA$ partially changed. Because control resumes at line 560, this value remains unpredictable:

```
550     STRINGA$ = "ABC"
560     PRINT #2%, STRINGA$
                !.
                !.
                !.
19000   PRINT "BUFFER POSITION IS ";CCPOS(2%)
19010   RESUME 560
```

To disable control/C trapping, use the RCTRLC function. For example:

```
800     Y% = CTRLC
                !.
                !.
                !.
1200    Y% = RCTRLC
                !.
                !.
                !.
19000   IF (ERR = 28%)
        THEN
                PRINT "CURRENT RECORD IS "; I%
                Y% = CTRLC
        ELSE
                ON ERROR GOTO 0
        END IF
19010   INPUT "DO YOU WISH TO END PROCESSING"; ANSWER$
19020   IF ANSWER$ = "YES"
        THEN
                RESUME 32000
        ELSE RESUME
        END IF
32000   CLOSE #1%
```

Note that the RCTRLC function only disables CTRL/C trapping, not the CTRL/C interrupts themselves.

11.8 The BASIC Error Handler

If a program that you compile, link, and run generates an error and has no routine to process it, BASIC handles the error. For most errors:

1. Execution halts.
2. The exit handler closes all files.
3. BASIC prints a message describing the nature of the error and optional system traceback information.
4. Control returns to monitor level.

Figure 11–2 shows the error messages and traceback generated by a VAX–11 BASIC program that has been compiled, linked, and run.

Figure 11–2: VAX–11 BASIC Error Handling Output

```
100     GOSUB 400
200     GOTO 32767
400     A% = A%/B%
500     RETURN
32767   END

$ RUN NONAME
%BAS-F-DIVBY_ZER, Division by 0
-BAS-I-USEPC_PSL,  at user PC=00000436, PSL=03C00026
-BAS-I-FROLINGSB,  from line 400 in GOSUB 400 in module NONAME
-BAS-I-FROLINMOD,  from line 100 in module NONAME
-SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at
PC=00004933, PSL=03C00000
%TRACE-F-TRACEBACK, symbolic stack dump follows
```

(continued on next page)

```
module name routine name            line     relative PC  absolute PC

                                                00004933     00004933
                                                80000014     80000014
                                                80000014     80000014
NONAME       NONAME$MAIN            400         00000025     00000425
$
```

Figure 11–3 shows the error messages and traceback generated by a VAX–11 BASIC program that has been compiled, linked, and run.

Figure 11–3: BASIC–PLUS–2 Error Handling Output

```
100      GOSUB 400
200      GOTO 32767
400      A% = A%/B%
500      RETURN
32767    END
$ RUN NONAME
%Division by 0 at line     400  in "NONAME"
```

If a program you run in the BASIC environment generates an error and includes no routine to process it, BASIC handles the error by performing these slightly different steps:

1. Execution halts.
2. The exit handler closes all files and disconnects all devices.
3. BASIC prints a message describing the nature of the error.
4. Control returns to BASIC, which displays its prompt.

Figure 11–4 shows the error messages generated by a program run in the BASIC environment on VAX/VMS systems.

Figure 11–4: VAX–11 BASIC Environment Error Handling Output

```
100 GOSUB 400
200 GOTO 32767
400 A% = A%/B%
500 RETURN
32767 END
RUN
NONAME          3-APR-1981 11:03
%BAS-F-DIVBY_ZER, Division by 0
-BAS-I-USEPC_PSL,  at user PC=00116506, PSL=03C00026
-BAS-I-FROLINGSB,  from line 400 in GOSUB 400 in module NONAME
-BAS-I-FROLINMOD,  from line 100 in module NONAME
-SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at
PC=0009017E, PSL=03C00000
Ready
```

Figure 11–5 shows the error messages generated by a program run in the BASIC environment on PDP–11 systems.

Figure 11–5: BASIC–PLUS–2 Environment Error Handling Output

```
100       GOSUB 400
200       GOTO 32767
400       A% = A%/B%
500       RETURN
32767     END
RUNNH
%Division by 0 at line     400 in "NONAME"
```

11.9 Handling Non–BASIC Errors

A user-written BASIC error handler cannot trap errors occurring in a non–BASIC subprogram. If a non–BASIC error reaches a BASIC main program, then the BASIC error handler terminates the program. However, if the main program is written in VAX–11 MACRO or VAX–11 BLISS, then a user-written error handler in the main program can get control and handle the error.

For example, suppose you have a VAX–11 MACRO main program that calls a BASIC subprogram that then calls a VAX–11 FORTRAN subprogram. Further, all error handling is done in the VAX–11 MACRO main program. An error in the VAX–11 FORTRAN subprogram is signaled to the BASIC subprogram. Because the BASIC subprogram cannot handle non–BASIC errors, it resignals the error to the VAX–11 MACRO main program. To handle the error without terminating the program, the VAX–11 MACRO main program must have a user-written error handler. See the *Guide to Creating Modular Library Procedures* Manual for more information.

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 conventions are used:

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.

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

BASIC
User's Guide
AA–L335A–TK
AD–L335A–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