HP BASIC for OpenVMS

User Manual

Order Number: AA–HY15F–TK

January 2005

This manual describes how to develop HP BASIC programs and use HP BASIC features on HP OpenVMS Industry Standard 64 and HP OpenVMS Alpha systems.

Revision/Update Information:	This revised manual supersedes the *Compaq BASIC OpenVMS Alpha and VAX Systems User Manual*, Version 1.4.
Software Version:	HP BASIC Version 1.6 for OpenVMS Systems
Operating System and Version:	OpenVMS I64 Version 8.2 or higher OpenVMS Alpha Version 7.1 or higher (with IEEE floating-point support) OpenVMS Alpha Version 6.1 or higher (without IEEE floating-point support)

Hewlett-Packard Company
Palo Alto, California

© Copyright 2005 Hewlett-Packard Development Company, L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Intel and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Printed in the US

ZK5424

This manual is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

3 Using the OpenVMS Debugger with BASIC

Part II Compaq BASIC Programming Concepts

4 BASIC Concepts and Elements

5 Simple Input and Output

6 Arrays

9 Program Control

10 Functions

11 String Handling

12 Program Segmentation

13 File Input and Output

14 Formatting Output with the PRINT USING Statement

15 Handling Run-Time Errors

16 Compiler Directives

17 Data Representation

Part III Using HP BASIC Features on OpenVMS Systems

18 Advanced File Input and Output

19 Using BASIC in the Common Language Environment

20 Libraries and Shareable Images

21 Using CDD/Repository with BASIC

22 Using DECwindows Motif Bindings with BASIC

A Compile-Time Error Messages

B Run-Time Messages

C Optional Programming Productivity Tools

Index

Examples

Figures

Tables

Preface

This manual describes how to develop and use HP BASIC programs on OpenVMS systems and describes BASIC language features.

Note

In this manual, the term OpenVMS refers to both OpenVMS I64 and OpenVMS Alpha systems. If there are differences in the behavior of the HP BASIC compiler on the two operating systems, those differences are noted.

The term I64 BASIC refers to HP BASIC on OpenVMS I64 systems.

Alpha BASIC refers to HP BASIC on OpenVMS Alpha systems.

VAX BASIC refers to VAX BASIC on OpenVMS VAX systems.

Intended Audience

This manual is intended for programmers who compile, link, and execute HP BASIC programs on OpenVMS systems. Users should have a working knowledge of BASIC or another high-level programming language, the Digital Command Language (DCL), and DCL command procedures.

Document Structure

This manual contains the following chapters and appendixes:

Part I Developing HP BASIC Programs on OpenVMS Systems

- Chapter 1 provides a brief overview of HP BASIC.
- Chapter 2 describes how to develop programs at DCL command level and how to generate a compiler listing.
- Chapter 3 describes how to use the OpenVMS Debugger to debug HP BASIC programs.

Part II HP BASIC Programming Concepts

- Chapter 4 explains how to get started with HP BASIC.
- Chapter 5 explains simple input and output procedures.
- Chapter 6 shows how to use arrays.
- Chapter 7 explains data definitions.
- Chapter 8 explains how to create user-defined data structures with the RECORD statement.
- Chapter 9 shows how to control the flow of program execution.
- Chapter 10 explains how to use functions.
- Chapter 11 explains how to handle strings.
- Chapter 12 describes structured programming techniques.
- Chapter 13 explains how to manage files.
- Chapter 14 describes how to format output with the PRINT USING statement.
- Chapter 15 explains error-handling techniques.
- Chapter 16 shows how to use compiler directives.
- Chapter 17 describes how BASIC represents data.

Part III Using HP BASIC Features on OpenVMS Systems

- Chapter 18 describes additional I/O considerations on OpenVMS systems.
- Chapter 19 describes OpenVMS System Services and Run-Time Library routines.
- Chapter 20 describes the use of user-supplied libraries and shareable images.
- Chapter 21 describes how to use CDD/Repository capabilities.
- Chapter 22 describes using standard Motif Bindings with BASIC.

Appendixes

- Appendix A lists compile-time error messages.
- Appendix B lists run-time error messages.
- Appendix C provides an overview of the optional productivity tools.

Related Documents

For more information about language elements, syntax, and reference information, see the *HP BASIC for OpenVMS Reference Manual*.

Reader's Comments

HP welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet	**openvmsdoc@hp.com**
Postal Mail	Hewlett-Packard Company OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

Conventions

The following product names may appear in this manual:

- HP OpenVMS Industry Standard 64 for Integrity Servers
- OpenVMS I64
- I64

All three names—the longer form and the two abbreviated forms—refer to the version of the OpenVMS operating system that runs on the Intel® Itanium® architecture.

The following typographic conventions might be used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.

. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.

-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Mnemonics and Other Terms Used in Syntax Diagrams

angle	Angle in radians or degrees
array	Array; syntax rules specify whether the bounds or dimensions can be specified
chnl	I/O channel associated with a file
chnl-exp	Numeric expression that specifies a channel number
com	Specific to a COMMON block
cond	Conditional expression; indicates that an expression can be either logical or relational
cond-exp	Conditional expression
const	Constant value
data-type	Data type keyword
decimal-var	Decimal variable
decl-item	Array, record, or variable
def	Specific to a DEF function
delim	Delimiter
equiv-name	File specification, device, or logical name to be assigned a logical name
err-num	Run-time error number
exp	Expression
ext-routine	External function
external-param	External parameter
file-spec	File specification
func	Specific to a FUNCTION subprogram
int	Integer value
int-const	Integer constant
int-exp	Expression that represents an integer value
int-var	Variable that contains an integer value

label	Alphanumeric statement label
lex	Lexical; used to indicate a component of a compiler directive
lex-exp	Lexical expression
lex-var	Lexical variable
line	Statement line; may or may not be numbered
line-num	Statement line number
lit	Literal value, in quotation marks
log-exp	Logical expression
log-name	1- to 63-character logical name to be associated with *equiv-name*
macro-id	User identifier following the rules for BASIC identifiers
map	Specific to a MAP statement
matrix	Two-dimensional array
name	Name or identifier; indicates the declaration of a name or the name of a BASIC structure, such as a SUB subprogram
num	Numeric value
num-lit	Numeric literal
param-list	Parameter list, such as for a SUB subprogram
pass-mech	Valid BASIC passing mechanism
prog-name	Program name
real	Floating-point value
real-exp	Real expression
real-var	Real variable
rec-exp	Record expression; record number within a file
rel-exp	Relational expression
relationship-type	Oracle CDD/Repository protocol
replacement-token	Identifier, keyword, compiler directive, literal constant, or operator
routine	SUB subprogram or other callable procedure
str	Character string
str-exp	Expression that represents a character string
str-lit	String literal
str-var	Variable that contains a character string
sub	Specific to a SUB subprogram

target	Target point of a branch statement; either a line number or a label
unq-str	Unique string
unsubs-var	Unsubscripted variable, as opposed to an array element
var	Variable

Part I

Developing BASIC Programs on OpenVMS Systems

Part I provides an overview of BASIC and describes how to develop and debug BASIC programs. It shows you how to get started on the OpenVMS system and how to develop programs both at DCL command level and within the VAX BASIC Environment.

1
Overview of HP BASIC

BASIC is a powerful structured programming language designed for novice and application programmers alike.

BASIC was originally developed for students with little or no programming experience. Since then, BASIC has become one of the most widely used programming languages and is available on almost every computer system.

The OpenVMS implementations of BASIC have evolved beyond the original design but still support all of the traditional features of the original language in addition to more recent programming techniques. HP BASIC has become much more than a teaching tool and is used in a wide variety of sophisticated applications.

1.1 Language Constructs Supported

HP BASIC supports the following language constructs:

- Code without line numbers (traditional line numbers are optional)
- Control structures, such as SELECT CASE
- Explicit variable declarations
- Capabilities for handling dynamic strings
- Adaptable file-handling capabilities for terminal-format files, and the full range of RMS facilities
- Global and local run-time error handling with WHEN ERROR blocks
- Compile-time directives
- A variety of data types, including packed-decimal, user-defined records, and VAX and IEEE floating-point data types.
- Extensive error checking with meaningful error messages
- Thirty-one character names for variables, labels, functions, and subprograms

1.2 Advantages on OpenVMS

HP BASIC uses the OpenVMS operating system to its full advantage and is integrated with many other HP products. In particular, HP BASIC supports:

- The OpenVMS systems standard calling procedures
- Record definitions included from the OpenVMS Common Data Dictionary
- Code analysis with the Performance and Coverage Analyzer (PCA)
- Creation of code with the Language-Sensitive Editor (LSE)
- Extensive online language help
- Exchange of data with other systems using DECnet

HP BASIC supports features of other versions of BASIC, including PDP-11 BASIC-PLUS-2. The /FLAG qualifier allows you to check whether programs contain declining features that should be replaced with newer ones.

When you write programs in HP BASIC, you develop programs at the DCL command level. You write your source program with a text editor, then compile, link, and run the program with commands to the OpenVMS operating system.

2
Developing HP BASIC Programs

This chapter describes how to compile, link, and run an HP BASIC program.

For information about using a text editor to create and edit files, see the *OpenVMS User's Manual*.

2.1 Compiling an HP BASIC Program

The HP BASIC compiler performs the following functions:

- Detects errors in your source program
- Generates any appropriate error messages
- Generates machine language instructions from the source statements
- Groups these language instructions into an object module for the linker

To invoke the compiler, you use the DCL command BASIC. With the BASIC command, you can specify command qualifiers. The next sections discuss the BASIC command in detail as well as the command qualifiers available.

2.1.1 BASIC Command

When you compile your source program, use the BASIC command, which has the following format:

BASIC [*/qualifier...*][*file-spec* [*/qualifier...*]],...

/qualifier
Indicates a specific action to be performed by the compiler on all files or specific files listed. When a qualifier appears directly after the BASIC command, it affects all files listed.

file-spec
Indicates the name of the input source file that contains the program or module to be compiled. You are not required to specify a file extension; the HP BASIC compiler assumes the default file type .BAS.

Most of the command qualifiers to the BASIC command affect all files specified in the command line, no matter where the qualifiers are placed; these are called **global qualifiers**. However, the qualifiers /LISTING, /OBJECT, /DIAGNOSTICS, and /ANALYSIS_DATA are **positional qualifiers**; that is, depending on their position in the command line, they can affect all or only some of the specified files. The rules for positional qualifiers are as follows:

- If the positional qualifier is located directly following the command name, it affects all the specified files.
- If the file specifications are separated by commas, then any positional qualifier directly following a file specification affects only that file.
- If the file specifications are separated by plus signs, then any positional qualifier directly following a list of file specifications affects only the resulting appended file.
- The rightmost qualifier overrides any conflicting qualifier previously specified in the command line.

The placement of these positional qualifiers causes BASIC to produce or not produce listing files, object files, and diagnostics files. For example:

```
$ BASIC/LIST/OBJ PROG1/NOOBJ/DIAG,PROG2+PROG3/NOLIST
```

This command does the following:

- Compiles PROG1 and produces a listing file called PROG1.LIS
- Produces no object file for PROG1
- Produces a diagnostics file for PROG1 called PROG1.DIA
- Appends PROG2 and PROG3 for compilation, producing a temporary source file called PROG2
- Compiles the new PROG2 and produces an object file called PROG2.OBJ
- Produces no listing file for the new PROG2

HP BASIC does not require line numbers in either of the source files. The "+" operator is treated as an OpenVMS append operator. HP BASIC appends and compiles the separate files as if they were a single source file.

2.1.2 BASIC Command Qualifiers

The following list shows the BASIC command qualifiers and their defaults. A description of each qualifier follows the list.

The qualifiers that are "declining features" and no longer recommended are separately described in Section 2.1.3.

Command Qualifier	Default
/[NO]ANALYSIS_DATA [= *file-specification*]	/NOANALYSIS_DATA
/ARCHITECTURE = *arch-type*	/ARCHITECTURE = GENERIC
/[NO]AUDIT [= *text-entry*]	/NOAUDIT
/[NO]CHECK [= (*check-clause*,...)]	/CHECK = (BOUNDS,OVERFLOW)
/[NO]CROSS_REF [= [NO]KEYWORDS]	/NOCROSS_REF
/[NO]DEBUG [= (*debug-clause*,...)]	/DEBUG = (TRACEBACK,SYMBOLS)
/DECIMAL_SIZE = (d,s)	/DECIMAL_SIZE = (15,2)
/[NO]DEPENDENCY_DATA	/NODEPENDENCY_DATA
/[NO]DIAGNOSTICS [= *file-specification*]	/NODIAGNOSTICS
/[NO]FLAG [= *flag-clause*]	/FLAG = NODECLINING
/INTEGER_SIZE = *data-type*	/INTEGER_SIZE = LONG
/[NO]LINES	/NOLINES
/[NO]LISTING [= *file-specification*]	/NOLISTING (from terminal) /LISTING (batch)
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OBJECT [= *file-specification*]	/OBJECT
/[NO]OLD_VERSION [= CDD_ARRAYS]	/NOOLD_VERSION
/[NO]OPTIMIZE [= LEVEL = *n*]	/OPTIMIZE = LEVEL = 4
/REAL_SIZE = *data-type*	/REAL_SIZE = SFLOAT (I64)or SINGLE (Alpha)
/[NO]ROUND_DECIMAL	/NOROUND_DECIMAL
/SCALE = *n*	/SCALE = 0
/[NO]SEPARATE_COMPILATION	/NOSEPARATE_COMPILATION
/[NO]SHOW [= (*show-item*,...)]	/SHOW
/[NO]SYNCHRONOUS_EXCEPTIONS	/NOSYNCHRONOUS_EXCEPTIONS
/TYPE_DEFAULT = *default-clause*	/TYPE_DEFAULT = REAL
/VARIANT = *int-const*	/VARIANT = 0
/[NO]WARNINGS [= (*warn-clause*,...)]	/WARNINGS = (INFORMATIONALS, WARNINGS, NOALIGNMENT)

/[NO]ANALYSIS_DATA [= *file-specification*]

/NOANALYSIS_DATA (default)
The /ANALYSIS_DATA qualifier generates a file containing data analysis information. This file has the file type .ANA. The Source Code Analyzer (SCA) library uses these files to display cross-reference information and to analyze source code.

Remarks

- SCA must be installed.

/ARCHITECTURE [= { **GENERIC** | **HOST** | **EV4 (Alpha only)** | **EV5 (Alpha only)** | **EV56 (Alpha only)** | **PCA56 (Alpha only)** | **EV6 (Alpha only)** | **EV67 (Alpha only)** | **ITANIUM2 (I64 only)** | **MERCED (I64 only)** } **]**

/ARCHITECTURE = GENERIC (default)
The /ARCHITECTURE qualifier specifies which version of the Itanium or Alpha architecture to generate instructions for.

All Itanium and Alpha processors implement a core set of instructions and, in some cases, the following extensions: BWX (byte- and word-manipulation instructions) and MAX (multimedia instructions).

OpenVMS Version 7.1 and subsequent releases include an instruction emulator. This capability allows any Itanium or Alpha chip to execute and produce correct results from Itanium or Alpha instructions, respectively, even if some of the instructions are not implemented on the chip. Applications using emulated instructions will run correctly, but might incur significant emulation overhead at run time.

Remarks

- /ARCHITECTURE = GENERIC (default) generates instructions that are appropriate for all Itanium or Alpha processors.
- /ARCHITECTURE = HOST generates instructions for the Itanium or Alpha processor that the compiler is running on (for example, EV56 instructions on an EV56 processor, and EV4 instructions on an EV4 processor).
- /ARCHITECTURE = EV4 generates instructions for the EV4 processor (21064, 21064A, 21066, and 21068 Alpha chips).

  Programs compiled with this option will not incur any emulation overhead on any Alpha processor.
- /ARCHITECTURE = EV5 generates instructions for the EV5 processor (some 21164 Alpha chips).

Programs compiled with this option will not incur any emulation overhead on any Alpha processor.

- /ARCHITECTURE = EV56 generates instructions for the EV56 processor (some 21164 Alpha chips). This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX extension.

  Programs compiled with this option might incur emulation overhead on EV4 and EV5 processors.

  Note that the EV5 and EV56 processor both have the same chip number: 21164.

- /ARCHITECTURE = PCA56 generates instructions for the PCA56 processor (21164PC Alpha chip). This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX extension. Note that currently HP BASIC does not generate any of the instructions in the MAX extension to the Alpha architecture.

  Programs compiled with this option might incur emulation overhead on EV4 and EV5 processors.

- /ARCHITECTURE = EV6 generates instructions for the EV6 processor (21264 Alpha chip). This option permits the compiler to generate any EV4 instruction, any instructions contained in the BWX and MAX extensions, plus any instructions added for the EV6 chip. These instructions include a floating-point square root instruction (SQRT), integer/floating-point register transfer instructions, and additional instructions to identify extensions and processor groups.

  Programs compiled with this option might incur emulation overhead on EV4, EV5, EV56, and PCA56 processors.

- /ARCHITECTURE = EV67 generates instructions for the EV67 processor (21264A Alpha chip). This option permits the compiler to generate any EV6 instruction, plus bit count instructions (CTLZ, CTPOP, and CTTZ). However, HP BASIC does not generate any of the bit count instructions, so EV67 is essentially identical to EV6.

  Programs compiled with this option might incur emulation overhead on EV4, EV5, EV56, and PCA56 processors.

- /ARCHITECTURE = ITANIUM2 generates instructions for the Itanium 2 processor. This option permits the compiler to generate any Itanium 2 instructions.

- /ARCHITECTURE = MERCED generates instructions for the Merced processor. This option permits the compiler to generate any Merced instructions.

/[NO]AUDIT [= { *str-lit* / *file-specification* }]

/NOAUDIT (default)
The /AUDIT qualifier causes the compiler to include a history entry in CDD/Repository when extracting a CDD/Repository definition. You can specify either a string literal or a file specification with the /AUDIT qualifier. If you specify a string literal, BASIC includes it as part of the history entry. If you specify a file specification, BASIC includes up to the first 64 lines of the specified file. When you specify /AUDIT, BASIC also includes the following information about the CDD/Repository record extraction in the history entry:

- The name of the program module making the extraction
- The time and date of the extraction
- A note that access was made by way of a BASIC program
- A note that the access was an extraction
- The username and UIC of the process accessing CDD/Repository

Remarks

- /NOAUDIT causes the compiler not to include a history entry in CDD/Repository when extracting a CDD/Repository definition.

/[NO]CHECK [= ({ [NO]BOUNDS / [NO]OVERFLOW [= ([NO]INTEGER, [NO]DECIMAL)] / ALL / NONE }, . . .)]

/CHECK = (BOUNDS,OVERFLOW) (default)
The /CHECK qualifier causes the compiler to test for arithmetic overflow and for array references outside array boundaries when the program executes.

Remarks

- In Alpha BASIC, specifying /CHECK = NOBOUNDS causes bounds checking not to be performed on array parameters received by descriptor.
- /CHECK = NOBOUNDS should only be used for thoroughly debugged programs and when execution time is critical. The program is smaller and runs faster, but no error is signaled for an array reference outside the array boundaries. The program might get a memory management or access violation error at run time.

- /CHECK = OVERFLOW enables checking for integers and packed decimal numbers.
- /CHECK = NOOVERFLOW disables overflow checking.
- /NOCHECK causes the compiler not to test for arithmetic overflow or for array references outside array boundaries when the program executes.
- /CHECK = ALL is the same as /CHECK = (BOUNDS, OVERFLOW).
- /CHECK = NONE is the same as /NOCHECK.

/[NO]CROSS_REFERENCE [= [NO]KEYWORDS]

/NOCROSS_REFERENCE (default)
The /CROSS_REFERENCE qualifier causes the compiler to generate a cross-reference listing. The cross-reference list shows program symbols, classes, and the program lines in which they are referenced.

Remarks

- /CROSS_REFERENCE = KEYWORDS specifies that the cross-reference listing includes all references to BASIC keywords. In Alpha BASIC, if the /LIST qualifier is not specified as well, /CROSS_REFERENCE is ignored.
- The default for /CROSS_REFERENCE is NOKEYWORDS. See Chapter 16 for more information about cross-reference listings.
- /NOCROSS_REFERENCE specifies that no cross-reference listing be produced.

/[NO]DEBUG [= ({ [NO]SYMBOLS | [NO]TRACEBACK | ALL | NONE } , . . .)]

/DEBUG = (TRACEBACK,SYMBOLS) (default)
The /DEBUG qualifier causes the compiler to provide information for the OpenVMS Debugger and the system run-time error traceback mechanism. Neither TRACEBACK nor SYMBOLS affects a program's executable code. For more information about debugging, see Chapter 3.

Remarks

- /NODEBUG causes the compiler to suppress information for the OpenVMS Debugger and the system run-time error traceback mechanism.
- /DEBUG = ALL is the same as /DEBUG = (TRACEBACK,SYMBOLS).
- /DEBUG = NONE is the same as /NODEBUG.

/DECIMAL_SIZE = (*d,s*)

/DECIMAL_SIZE = (15,2) (default)
The /DECIMAL_SIZE qualifier lets you specify the default size for packed decimal data. You specify the total number of digits in the number and the number of digits to the right of the decimal point.

/DECIMAL_SIZE = (15,2) is the default. This default decimal size applies to all decimal variables for which the total number of digits and digits to the right of the decimal point are not explicitly declared. See the *HP BASIC for OpenVMS Reference Manual* for more information about packed decimal numbers.

/[NO]DEPENDENCY_DATA

/NODEPENDENCY_DATA (default)
The /DEPENDENCY_DATA qualifier generates a compiled module entity in the CDD$DEFAULT for each compilation unit.

Remarks

- A compiled module entity is generated only if CDD/Plus Version 4.0 or higher or CDD/Repository Version 5.0 or higher is installed on your system and if your current CDD$DEFAULT is a CDO-format dictionary.
- You must specify this qualifier if you want %INCLUDE %FROM %CDD and %REPORT %DEPENDENCY directives to establish dependency relationships.
- /NODEPENDENCY_DATA causes the compiler not to generate a compiled module entity.

/[NO]DIAGNOSTICS [= *file-spec*]

/NODIAGNOSTICS (default)
The /DIAGNOSTICS qualifier creates a diagnostics file containing compiler messages and diagnostic information. The diagnostics file is used by LSE to display diagnostic error messages and to position the cursor on the line and column where a source error exists.

Remarks

- The Language-Sensitive Editor (LSE) must be installed.
- If you do not supply a file specification with the /DIAGNOSTICS qualifier, the diagnostics file has the same name as its corresponding source file and the file type .DIA. All other file specification attributes depend on the placement of the qualifier in the command. See the OpenVMS documentation set for more information.
- /NODIAGNOSTICS specifies that no diagnostics file is created.

/[NO]FLAG [= { [NO]DECLINING | ALL | NONE }]

/FLAG = NODECLINING (default)
The /FLAG qualifier lets you specify whether BASIC warns you about declining features.

Remarks

- /NOFLAG causes the compiler to issue no warnings about declining features.
- /FLAG = ALL is the same as /FLAG = DECLINING.
- /FLAG = NONE is the same as /NOFLAG.

/INTEGER_SIZE = { BYTE | WORD | LONG | QUAD }

/INTEGER_SIZE = (LONG) (default)
The /INTEGER_SIZE qualifier lets you specify the default size for integer data.

Remarks

- The default integer size (LONG) applies to all integer variables whose data type is not explicitly declared. See the *HP BASIC for OpenVMS Reference Manual* for more information about integer data types.

/[NO]LINES

/NOLINES (default)
The /LINES qualifier makes line number information available for the ERL function and the BASIC error reporter.

Remarks

- /NOLINES causes line number information to be unavailable for the ERL function and the HP BASIC error reporter. Specifying /NOLINES makes your program run faster and reduces program size. However, specifying /NOLINES causes the following restrictions to be in effect:
  - You cannot use the ERL function.
  - No BASIC line number is given in run-time error messages.

/[NO]LISTING [= *file-spec*]

/LISTING (default in batch mode)

/NOLISTING (default in interactive mode)
The /LISTING qualifier causes BASIC to produce a source listing file.

Remarks

- /LISTING = *file-spec* produces a file with an explicit file specification. Omitting the *file-spec* produces a listing file with the same name as its corresponding source file and a file type of .LIS.
- All other file specification attributes depend on the placement of the qualifier in the command. See the *OpenVMS User's Manual* for more information.
- /LISTING only controls whether or not the compiler produces a listing file and is the default in batch mode.
- /SHOW controls which parts of the listing are produced.
- /NOLISTING specifies that no source listing file be produced and is the default at a terminal.

/[NO]MACHINE_CODE

/NOMACHINE_CODE (default)
The /MACHINE_CODE qualifier specifies that the listing file includes the compiler-generated object code.

Remarks

- /MACHINE_CODE specifies that the compiler include a listing of the compiler-generated object code in the listing file. If the /LISTING qualifier is not specified as well, /MACHINE is ignored.
- /NOMACHINE_CODE specifies that the listing file not include compiler-generated object code.

/[NO]OBJECT [= *file-spec*]

/OBJECT (default)
The /OBJECT qualifier causes the compiler to produce an object module and optionally specifies its file name. By default, the compiler generates object files as follows:

- If you specify one source file, BASIC generates one object file.
- If you specify multiple source files separated by plus signs (+), BASIC appends the files and generates one object file.
- If you specify multiple source files separated by commas (,), BASIC compiles and generates a separate object file for each source file.
- You can use both plus signs and commas in the same command line to produce different combinations of appended and separated object files.

Remarks

- /OBJECT = *file-spec* produces an object file with an explicit file specification. Omitting *file-spec* causes the compiler to produce an object file having the same name as its corresponding source file and the file type .OBJ. All other file specification attributes depend on the placement of the qualifier in the command. See the *OpenVMS User's Manual* for more information.
- /NOOBJECT suppresses the creation of an object file. During the early stages of program development, you might find it helpful to suppress the production of object files until your source program compiles without errors.

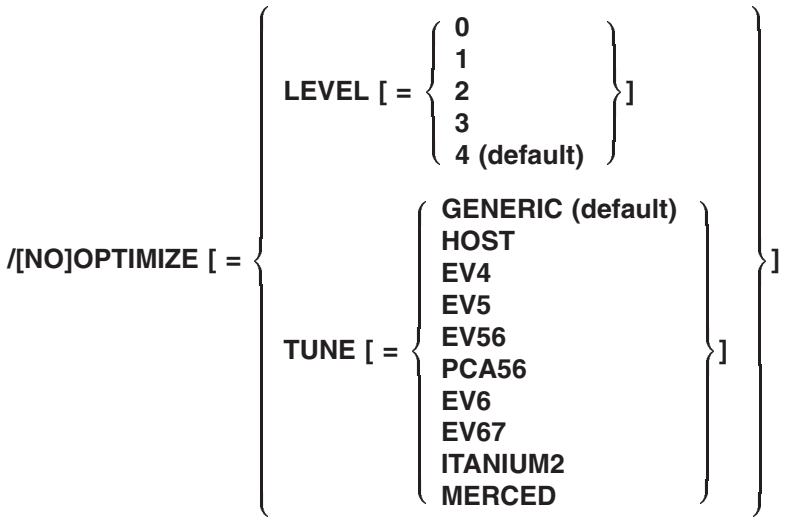
/[NO]OLD_VERSION [= CDD_ARRAYS]

/NOOLD_VERSION (default)
The /OLD_VERSION qualifier causes the compiler to change the lower bound to zero and adjusts the upper bound of the array. For example, *Array 2:5* in CDD/Repository is translated by the compiler to be an array with a lower bound of 0 and an upper bound of 3. The compiler issues an informational message to confirm the array bounds.

The /NOOLD_VERSION qualifier causes the compiler to extract an array from the CDD/Repository with the bounds as specified in the data definition. For example, *Array 2:5* in CDD/Repository is translated by the compiler to be an array with a lower bound of 2 and an upper bound of 5.

Remarks

- /OLD_VERSION [= CDD_ARRAYS] is provided for compatibility with previous versions of BASIC.
- CDD/Repository assumes a default lower bound of 1, if none is specified. Therefore, if no lower bound is specified, the compiler translates the CDD/Repository array to have a lower bound of 1. For example, *Array 5* in CDD/Repository is translated by HP BASIC to be an array with a lower bound of 1 and an upper bound of 5.



/OPTIMIZE = LEVEL = 4 (default)

/OPTIMIZE = TUNE = GENERIC (default)
The /OPTIMIZE qualifier causes the compiler to optimize the program to generate more efficient code for optimum run-time performance. Specifying /NOOPTIMIZE causes the compiler to perform minimal optimizations.

The following list describes the /OPTIMIZE = LEVEL options:

- 0 has the same effect as /NOOPTIMIZE. Most optimizations are turned off.
- 1 has some optimizations (such as instruction scheduling).

- 2 adds more optimizations (such as loop unrolling and split lifetime analysis) to those in level 1.
- 3 adds more optimizations.
- 4 is the default level.

  /OPTIMIZE = LEVEL = 4 is equivalent to /OPTIMIZE or not specifying the qualifier. Level 4 is the maximum optimization level.

The /OPTIMIZE = TUNE qualifier selects processor-specific instruction tuning for a specific implementation of the Itanium or Alpha architecture. Tuning for a specific implementation can provide improvements in run-time performance.

Regardless of the setting of the /OPTIMIZE = TUNE qualifier, the generated code will run correctly on all implementations of the Itanium or Alpha architecture as appropriate. Note that code tuned for a specific target might run more slowly on another target than generically-tuned code.

The following list describes the /OPTIMIZE = TUNE options:

- GENERIC (default) selects instruction tuning that is appropriate for all implementations of the Itanium or Alpha architecture.
- HOST selects instruction tuning that is appropriate for the Itanium or Alpha machine on which the code is being compiled.
- EV4 selects instruction tuning for the 21064, 21064A, 21066, and 21068 implementation of the Alpha architecture.
- EV5 selects instruction tuning for the 21164 implementation of the Alpha architecture.
- EV56 selects instruction tuning for the 21164 implementation of the Alpha architecture.
- PCA56 selects instruction tuning for the 21164PC implementation of the Alpha architecture.
- EV6 selects instruction tuning for the 21264 implementation of the Alpha architecture.
- EV67 selects instruction tuning for the 21264A implementation of the Alpha architecture.
- ITANIUM2 selects instruction tuning for the Itanium 2 implementation of the Itanium architecture.
- MERCED selects instruction tuning for the Merced implementation of the Itanium architecture.

Remarks

- Specify /NOOPTIMIZE if you specify /DEBUG when compiling a program. /NOOPTIMIZE expedites and simplifies the debugging session by putting the machine code in the same order as the lines in the source program. Optimizations can cause unexpected and confusing behavior in a debugging session.
- Specifying /OPTIMIZE, the default, usually makes programs run faster. However, using /OPTIMIZE produces extra instructions to perform the optimization, which might result in larger object modules and longer compile times than the /NOOPTIMIZE qualifier.
- To speed compilations during program development, compile with /NOOBJECT qualifier to check syntax, with /NOOPTIMIZE to check for correct execution, and finally with /OPTIMIZE for the final check.

/REAL_SIZE = { **SINGLE** | **DOUBLE** | **GFLOAT** | **SFLOAT** | **TFLOAT** | **XFLOAT** }

/REAL_SIZE = SFLOAT (I64 default); SINGLE (Alpha default)
The /REAL_SIZE qualifier specifies the default size for floating-point data.

Remarks

- The default floating-point size applies to all floating-point variables whose size is not explicitly declared.

See the *HP BASIC for OpenVMS Reference Manual* for more information about floating-point data types.

/[NO]ROUND_DECIMAL

/NOROUND_DECIMAL (default)
The /ROUND_DECIMAL qualifier causes the compiler to round packed decimal numbers rather than truncate them.

The /NOROUND_DECIMAL qualifier causes the compiler to truncate packed decimal numbers rather than round them.

The /ROUND_DECIMAL qualifier causes the INTEGER function to round rather than truncate the decimal part.

/SCALE = *n*

/SCALE = 0 (default)
The /SCALE qualifier specifies a scale factor from zero to six, inclusive. The scale factor affects only double-precision numbers. The SCALE qualifier helps to control accumulated round-off errors by multiplying floating-point values by 10 raised to the scale factor before storing them in variables. It is ignored for all but VAX double-precision (DOUBLE) floating-point numbers.

Remarks
The /SCALE qualifier is provided for compatibility with existing programs and with other implementations of BASIC. It is recommended that you do not use this feature for new program development. Accumulated round-off errors can be better controlled with packed decimal numbers. See the *HP BASIC for OpenVMS Reference Manual* for more information about packed decimal numbers.

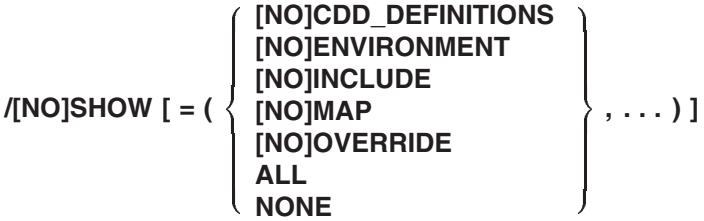
/[NO]SEPARATE_COMPILATION

/NOSEPARATE_COMPILATION (default)
The /SEPARATE_COMPILATION qualifier causes the compiler to place individual compilation units in separate modules in the object file. /NOSEPARATE_COMPILATION, the default, groups individual compilation units in a source file as a single module in the object file.

When creating modules for use in an object library, consider using /SEPARATE_COMPILATION to minimize the size of the routines included by the linker as it creates the executable image. /SEPARATE_COMPILATION also reduces the compiler virtual memory requirements when a source contains several compilation units.

Remarks

- /SEPARATE_COMPILATION causes the compiler to place each routine in a separate module within the output object.
- /NOSEPARATE_COMPILATION, in most cases, allows more interprocedural optimizations.



/SHOW = (CDD_DEFINITIONS, ENVIRONMENT, INCLUDE, MAP, NOOVERRIDE) (default)
The /SHOW qualifier determines which parts of the compilation listing are created.

Remarks

- The size value for dynamically mapped arrays is the size of the actual array.
- /LISTING must be specified for /SHOW to be effective.
- CDD_DEFINITIONS controls whether the translation of a CDD/Repository record is displayed in the listing.
- ENVIRONMENT lets you display all defaults that were in effect when the program was compiled. This is the compilation listing equivalent of the SHOW command in the environment.
- INCLUDE controls whether files accessed with the %INCLUDE directive are displayed in the listing.
- MAP determines whether the listing contains an allocation map. The allocation map lists all program variables, their size, and their data type.
- OVERRIDE helps you debug code by disabling the effect of the %NOLIST directive.
- /NOSHOW causes the compiler to display only the source listing.
- /SHOW = ALL is the same as /SHOW = (CDD_DEFINITIONS, ENVIRONMENT, INCLUDE,MAP, OVERRIDE).
- /SHOW = NONE is the same as /NOSHOW.

/[NO]SYNCHRONOUS_EXCEPTIONS

/NOSYNCHRONOUS_EXCEPTIONS (default)
The default /NOSYNCHRONOUS_EXCEPTIONS qualifier allows the compiler to reorder the execution of certain arithmetic instructions to improve performance on the hardware. If a program generates an arithmetic exception, such as an overflow or divide by zero, certain statements surrounding the offending statement may or may not be executed as a result of this reordering. Consider this example:

```
A = B
C = D / E
G = F
```

If the value of E is zero, the second statement will generate a divide by zero error. As a result of instruction reordering, it is possible that the assignment A = B will not take place. Further, it is possible that the assignment G = F will take place even though the previous statement generated an error.

The /SYNCHRONOUS_EXCEPTIONS qualifier disables reordering. Use this qualifier for programs that rely on arithmetic exceptions to occur at precise times during program execution.

The /SYNCHRONOUS_EXCEPTIONS qualifier impacts only arithmetic exceptions and variable assignments in the immediate area of the excepting statement.

Very few programs should require the /SYNCHRONOUS_EXCEPTIONS qualifier to produce correct results.

/TYPE_DEFAULT = { **INTEGER** | **REAL** | **DECIMAL** | **EXPLICIT** }

/TYPE_DEFAULT = REAL (default)
The /TYPE_DEFAULT qualifier lets you specify the default data type for numeric variables.

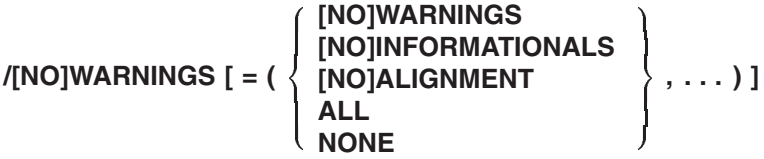
Remarks

- EXPLICIT specifies that all program variables must be explicitly declared in DECLARE, EXTERNAL, COMMON, MAP, or DIM statements.
- INTEGER, REAL, or DECIMAL specify that only variables and data which are not explicitly declared default to integer, real, or packed decimal.
- INTEGER_SIZE, REAL_SIZE, and DECIMAL_SIZE cause the compiler to specify the actual size of variables and data.

/VARIANT = *int-const*
The /VARIANT qualifier lets you specify the value associated with the lexical function %VARIANT. See Chapter 16 for more information about VARIANT and the %VARIANT lexical function.

Remarks

- If /VARIANT is not specified, the default value is 0.
- If /VARIANT is specified without a value, the default is 1.



/WARNINGS = (INFORMATIONAL,WARNINGS,NOALIGNMENT) (default)
The /WARNINGS qualifier lets you specify whether BASIC displays informational and warning messages.

Remarks

- /WARNINGS = NOWARNINGS causes the compiler to display informational messages but not warning messages.
- /WARNINGS = NOINFORMATIONALS causes the compiler to display warning messages but not informational messages.
- /NOWARNINGS causes the compiler to suppress any informational or warning messages.
- /WARNINGS = ALIGNMENT causes the compiler to flag all occurrences of non-naturally aligned RECORD fields, variables within COMMONs and MAPs, and RECORD arrays.

  An aligned data item starts on an address that is natural for that data type. Unaligned data accesses on Alpha can significantly reduce performance. Table 2–1 lists the natural boundaries for the supported data types.

Table 2–1 Natural Boundaries For Supported Data Types

Data Type	Natural Boundary
BYTE	BYTE
DECIMAL	BYTE
DOUBLE	QUADWORD
DYNAMIC STRING	BYTE
GFLOAT	QUADWORD
LONG	LONGWORD
QUAD	QUADWORD
RECORD	Depends on contents
RFA	BYTE
SFLOAT	LONGWORD
SINGLE	LONGWORD
STATIC STRING	BYTE
TFLOAT	QUADWORD
WORD	WORD
XFLOAT	OCTAWORD

/WARNINGS = NOALIGNMENT, the default, causes the compiler not to issue any warning messages about unaligned data.

The compiler naturally aligns all local variables and arrays, but it is the responsibility of the BASIC programmer to naturally align COMMONs, MAPS, and RECORDs. The /WARNINGS = ALIGNMENT qualifier flags all occurrences of non-naturally aligned items. This helps the programmer identify and correct unaligned entities.

An entity can be unaligned in the following ways:

- The entity does not start on a natural boundary for its data type. There are several actions a programmer can take to resolve this:
  - Rearrange the RECORD, MAP, or COMMON so that all entities start on natural boundaries.
  - Force proper alignment with fill items, as needed.

  Note that the natural alignment for a RECORD is equal to the largest alignment required by any of its fields. As an example, if a RECORD has a byte, long, and double field, the alignment of the RECORD would be quadword.

- For arrays of RECORDs and GROUPs, items can be unaligned if the size of a RECORD or GROUP is not a multiple of the alignment requirements of that RECORD or GROUP. For example, if a RECORD has a natural alignment of quadword, the size of the RECORD must be a multiple of eight. Otherwise, all array elements after the first might start on an unaligned boundary. Avoid unaligned accesses by padding the end of the RECORD with fill items.

- /WARNINGS = ALL is the same as /WARNINGS = (INFORMATIONAL, WARNINGS, ALIGNMENT).
- /WARNINGS = NONE is the same as /NOWARNINGS.

2.1.3 Declining Qualifiers and Their Recommended Replacements

The following qualifiers are declining features:

/BYTE
/DOUBLE
/GFLOAT
/LONG
/SINGLE
/TIE
/WORD

It is recommended that you replace them with newer qualifiers, as follows:

Old Qualifier	Recommended Replacement
/BYTE	/INTEGER_SIZE = BYTE
/DOUBLE	/REAL_SIZE = DOUBLE
/GFLOAT	/REAL_SIZE = GFLOAT
/LONG	/INTEGER_SIZE = LONG
/SINGLE	/REAL_SIZE = SINGLE
/TIE	Move to using entirely native code
/WORD	/INTEGER_SIZE = WORD

See the description of the /[NO]FLAG = [NO]DECLINING qualifier in this chapter. Also see the descriptions of the /INTEGER_SIZE and /REAL_SIZE qualifiers in this chapter. The old qualifiers are described in the *HP BASIC for OpenVMS Reference Manual*.

2.1.4 Compiler Listings

A **compiler listing** provides information that can help you debug your HP BASIC program. To generate a listing file, specify the /LISTING qualifier when you compile your HP BASIC program interactively. For example:

```
$ BASIC/LISTING prog-name
```

If the program is compiled as a batch job, the listing file is created by default; specify the /NOLISTING qualifier to suppress creation of the listing file. By default, the name of the listing file is the name of the source program followed by the file type .LIS. You can include a file specification with the /LISTING qualifier to override this default.

A compiler listing generated by the /LISTING qualifier has the following major sections:

- Source Program Listing

  The source program section contains the source code and line numbers generated by the compiler.

- Cross Reference

  The cross reference section is present if the /CROSS_REFERENCE qualifier was specified. It contains cross references of variables, symbols, and so forth.

- Allocation Map

  The allocation map section contains summary information about program sections, variables, and arrays.

- Qualifier Summary

  The qualifier summary section lists the qualifiers used with the BASIC command and the compilation statistics.

- Machine Code

  The machine code section is present if the /MACHINE_CODE qualifier was specified. It contains a symbolic representation of the machine instructions generated for the program section.

2.2 Linking an HP BASIC Program

On OpenVMS systems, the OpenVMS Linker (linker) simplifies the job of each language compiler because the logic needed to resolve symbolic references need not be duplicated. The main advantage to a system that has a linker, however, is that individual program modules can be separately written and compiled, and then linked together. This includes object modules produced by different language compilers.

The linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference
- Produces an executable image

When you link a program in development, in order to enable debugging, use the /DEBUG qualifier with the LINK command. The /DEBUG qualifier appends to the image all the symbol and line number information appended to the object modules plus information about global symbols, and forces the image to run under debugger control when you execute it (unless you then specify /NODEBUG).

The LINK command produces an executable image by default; however, you can also use the LINK command to obtain shareable images and system images. The /SHAREABLE qualifier directs the linker to produce a shareable image; the /SYSTEM qualifier directs the linker to produce a system image. See Section 2.2.2 for a complete description of these and other LINK command qualifiers.

For a complete discussion of the OpenVMS Linker, see the *HP OpenVMS Linker Utility Manual*.

2.2.1 LINK Command

Once you have compiled your source program or module, you link it by using the DCL command LINK. The LINK command combines your object modules into one executable image, which can then be executed by the OpenVMS system. A source program or module cannot run on the OpenVMS system until it is linked. The format of the LINK command is as follows:

LINK[/*command-qualifier*]... {*file-spec* [/*file-qualifier*...]},...

/command-qualifier
Specifies one or more output file options.

file-spec
Specifies the input file or files to be linked.

/file-qualifier
Specifies one or more input file options.

If you specify more than one input file, you must separate the input file specifications with plus signs (+) or commas (,). By default, the linker creates an output file with the name of the first input file specified and the file type .EXE. When you link more than one file, list the file containing the main program first. This way, the name of your output file will have the same name as that of your main program module.

The following command line links the object files DANCE.OBJ, CHACHA.OBJ, and SWING.OBJ to produce one executable image called DANCE.EXE:

```
$ LINK DANCE.OBJ, CHACHA.OBJ, SWING.OBJ
```

2.2.2 LINK Command Qualifiers

The LINK command qualifiers can be used to modify linker output, as well as to invoke the debugging and traceback facilities. Linker output consists of an image file and an optional map file. Image file qualifiers, map file qualifiers, and debugging and traceback qualifiers are described in this section.

This section summarizes some of the most commonly used LINK command qualifiers. For a complete list and description of LINK qualifiers, see the *HP OpenVMS Linker Utility Manual*.

/BRIEF
The /BRIEF qualifier causes the linker to produce a summary of the image's characteristics and a list of contributing modules. This qualifier is used with /MAP.

/[NO]CROSS_REFERENCE

/NOCROSS_REFERENCE (default)
The /CROSS_REFERENCE qualifier causes the linker to produce cross-reference information for global symbols; the /NOCROSS_REFERENCE qualifier causes the linker to suppress cross-reference information.

/[NO]DEBUG

/NODEBUG (default)
The /DEBUG qualifier causes the linker to include the OpenVMS Debugger information in the executable image and generates a symbol table; the /NODEBUG qualifier causes the linker to prevent debugger control of the program. The default is /NODEBUG.

/[NO]EXECUTABLE [= *file-spec*]

/EXECUTABLE (default)
The /EXECUTABLE qualifier causes the linker to produce an executable image; the /NOEXECUTABLE qualifier suppresses production of an image file. If a *file-spec* is given, the resulting image is given the name of the *file-spec*.

/FULL
The /FULL qualifier causes the linker to produce a summary of the image's characteristics, a list of contributing modules, listings of global symbols by name and by value, and a summary of characteristics of image sections in the linked image. This qualifier is used with /MAP.

/[NO]MAP [= *file-spec*]

/NOMAP (default interactive mode)

/MAP (default batch mode)
The /MAP qualifier causes the linker to generate a map file; the /NOMAP qualifier suppresses the map. If a *file-spec* is given, the map file is given the name of the *file-spec*.

/[NO]SHAREABLE

/NOSHAREABLE (default)
The /SHAREABLE qualifier causes the linker to create a shareable image; the /NOSHAREABLE qualifier generates an executable image.

/[NO]TRACEBACK

/TRACEBACK (default)
The /TRACEBACK qualifier causes the linker to generate symbolic traceback information when error messages are produced; the /NOTRACEBACK qualifier suppresses traceback information.

2.2.3 Linker Input Files

You can specify the object modules to be included in an executable image in any of the following ways:

- Specify input file specifications for the object modules.

  If no file type is specified, the linker assumes that an input file is an object file with the file type .OBJ.

- Specify one or more object module library files.

  You can either specify the name of an object module library with the /LIBRARY qualifier, or specify the names of object modules contained in an object module library with the /INCLUDE qualifier. The uses of object module libraries are described in Section 2.2.5.

- Specify an options file.

  An options file can contain additional file specifications for the LINK command as well as special linker options. You must use the /OPTIONS qualifier to specify an options file. For more information about options files, see the *HP OpenVMS Linker Utility Manual*.

The linker uses the following default file types for input files:

File	File Type
Object module	.OBJ
Object library	.OLB
Options file	.OPT

2.2.4 Linker Output Files

When you enter the LINK command interactively and do not specify any qualifiers, the linker creates only an executable image file. By default, the resulting image file has the same file name as the first object module specified, and the file type .EXE.

In a batch job, the linker creates both an executable image file and a storage map file by default. The default file type for map files is .MAP.

To specify an alternative name for a map file or image file, or to specify an alternative output directory or device, you can include a file specification on the /MAP or /EXECUTABLE qualifier. For example:

```
$ LINK UPDATE/MAP=TEST
```

2.2.5 Using an Object Module Library

In a large development effort, the object modules for subprograms are often stored in an object module library. By using an object module library, you can make program modules contained in the library available to other programmers. To link modules contained in an object module library, use the /INCLUDE qualifier and specify the specific modules you want to link. For example:

```
$ LINK GARDEN, VEGGIES/INCLUDE = (EGGPLANT,TOMATO,BROCCOLI,ONION)
```

This example directs the linker to link the object modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main object module GARDEN.

Besides program modules, an object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which they are defined. You specify the name of the object module library containing symbol definitions with the /LIBRARY qualifier. When you use the /LIBRARY qualifier during a link operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

In the following example, the linker uses the library RACQUETS to resolve undefined symbols in BADMINTON, TENNIS, and RACQUETBALL:

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library, such as LNK$LIBRARY, to be your default library by using the DCL command DEFINE. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the LINK command. See the *HP OpenVMS DCL Dictionary* for more information about the DEFINE command.

For more information about object module libraries, see the *HP OpenVMS Linker Utility Manual*.

2.2.6 Linker Error Messages

If the linker detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any error or fatal error conditions occur (errors with severities of E or F), the linker does not produce an image file.

The messages produced by the linker are descriptive, and you do not usually need additional information to determine the specific error. Some common errors that occur during linking are as follows:

- An object module has compilation errors.

  This error occurs when you attempt to link a module that has warnings or errors during compilation. You can usually link compiled modules for which the compiler generated messages, but you should verify that the modules will actually produce the output you expect.

- The input file has a file type other than .OBJ and no file type was specified on the command line.

  If you do not specify a file type, the linker assumes the file has a file type of .OBJ by default. If the file is not an object file and you do not identify it with the appropriate file type, the linker signals an error message and does not produce an image file.

- You tried to link a nonexistent module.

  The linker signals an error message if you misspell a module name on the command line or if the compilation contains fatal diagnostics.

- A reference to a symbol name remains unresolved.

  An error occurs when you omit required module or library names from the command line and the linker cannot locate the definition for a specified global symbol reference. For example, a main program module OCEAN.OBJ calls the subprograms located in object modules REEF.OBJ, SHELLS.OBJ, and SEAWEED.OBJ. However, the following LINK command does not reference the object module SEAWEED.OBJ:

  ```
  $ LINK OCEAN, REEF, SHELLS
  ```

  This example produces the following error messages:

  ```
  %LINK-W-NUDFSYMS, 1 undefined symbol
  %LINK-I-UDFSYMS,        SEAWEED
  %LINK-W-USEUNDEF, module "OCEAN" references undefined symbol "SEAWEED"
  %LINK-W-DIAGISUED, completed but with diagnostics
  ```

If an error occurs when you link modules, you can often correct the error by reentering the command string and specifying the correct modules or libraries.

See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a complete list of linker messages.

2.3 Running an HP BASIC Program

After you link your program, use the DCL command RUN to execute it. The RUN command has the following format:

RUN [/[NO]DEBUG] *file-spec* [/[NO]DEBUG]

/[NO]DEBUG
The /[NO]DEBUG qualifier is optional. Specify the /DEBUG qualifier to request the debugger if the image is not linked with it. You cannot use /DEBUG on images linked with the /NOTRACEBACK qualifier. If the image is linked with the /DEBUG qualifier, and you do not want the debugger to prompt, use the /NODEBUG qualifier. The default action depends on whether the file is linked with the /DEBUG qualifier.

file-spec
The name of the file you want to execute.

The following example executes the image SAMPLE.EXE without invoking the debugger:

```
$ RUN SAMPLE/NODEBUG
```

See Chapter 3 for more information about debugging programs.

During program execution, an image can generate a fatal error called an **exception condition**. When an exception condition occurs, HP BASIC displays an error message. Run-time errors can also be issued by other facilities, such as the OpenVMS operating system. For more information about run-time errors, see Appendix B.

2.3.1 Improving Run-Time Performance of HP BASIC Programs

Even with fast hardware and an optimizing compiler, you can still tune your code for run-time performance. This section provides recommendations to consider if further performance improvements are desirable.

To achieve the best performance for your application, it is important to let both the hardware and the optimizer/code generator take advantage of their full capabilities. This can be accomplished by minimizing, and in some cases avoiding, the use of language features and qualifiers that block optimal program execution.

2.3.1.1 Data Items

Choose data types and align data items with the following in mind:

- Align data items in MAP, COMMON, and RECORD statements. This is the recommended first step to improve performance. For more information on alignment, see Section 2.1.2 under /WARNING = ALIGNMENT.
- Use LONG or QUAD data items instead of BYTE and WORD; accessing LONG or QUAD items is faster than BYTE and WORD, which may require multiple hardware instructions.
- On Alpha, use GFLOAT or TFLOAT data items instead of DOUBLE; operations are faster on GFLOAT and TFLOAT items. Operations on DOUBLE operands are performed by converting to GFLOAT, performing the operation in GFLOAT, and converting back to DOUBLE.
- On Itanium, use IEEE data items instead of VAX floating-point data items. VAX data type operands are converted to appropriate IEEE types before being operated on.
- Choose packed decimal lengths that are the most efficient while still meeting the needs of the application. The most efficient sizes are the default size of 15 digits (which fits exactly in a quadword) and 7 digits (which fits exactly in a longword). If you use one of these preferred sizes, it should be aligned on a quadword or longword boundary.
- Use packed decimal only when it is the appropriate data type. For example, do not use packed decimal to specify array subscripts, which are integers.
- Minimize mixed data type expressions, especially when you use packed decimal.

2.3.1.2 Qualifiers

On your BASIC command line, consider the following when you specify qualifiers:

- Use overflow and bounds checking only if they are needed. (See Section 2.1.2; bounds checking is needed if your program is not thoroughly debugged.) Both of these /CHECK options are on by default and will hinder performance.
- The use of the /LINES qualifier can impede optimization. /LINES is needed in Alpha BASIC only for the ERL function and to print BASIC line numbers in run-time error messages. /NOLINES is the default in Alpha BASIC.

- The default optimization level, /OPTIMIZATION = LEVEL = 4, provides the highest level of optimization.
- The /SYNCHRONOUS_EXCEPTIONS qualifier inhibits many optimizations. For more information on /SYNCHRONOUS_EXCEPTIONS, see Section 2.1.2.

2.3.1.3 Statements

The statements used in a program can affect performance, as follows:

- If you use error handling, the default ON ERROR GO BACK has the least impact on performance. ON ERROR GOTO {target} and WHEN blocks have a greater impact. If the application spends a large percentage of time in one routine, consider writing the routine with default error handling, if possible.
- RESUME without a target impedes optimization. (This applies only to RESUME statements that do not specify a target.)
- A MOVE TO or FIELD statement limits optimizations in the entire routine (SUB, FUNCTION, or main) where the statement is found. There is no additional cost for any statement after the first.
- OPTION INACTIVE = SETUP can dramatically minimize routine startup times by omitting RTL calls that initialize and close down routines. For small BASIC routines, the overhead of these RTL calls can be significant. Use this option for routines that are frequently called.

  If your routine contains any of the following elements, the compiler provides an informational diagnostic and emits calls to the RTL initialization and close-down routines:

  CHANGE statements
  DEF statements
  Dynamic string variables
  Executable DIM statements
  EXTERNAL string functions
  MAT statements
  MOVE statements for an entire array
  ON ERROR statements
  READ statements
  REMAP statements
  RESUME statements
  WHEN blocks
  String concatenation
  Built-in string functions
  Virtual arrays

Routines using OPTION INACTIVE = SETUP cannot perform I/O and have no error-handling capabilities. If an error occurs in such a routine, the error is resignaled to the calling routine.

Using OPTION INACTIVE = SETUP instructs the compiler not to emit code to initialize local variables. This also improves run-time performance, but impacts routines that rely upon the automatic initialization of local variables.

- CONTINUE without a target and RETRY can limit optimizations within the scope of the WHEN blocks associated with the handler that contains these statements. This impact can be significant if the handler is associated with a large WHEN block. The code within the associated WHEN blocks will be minimally optimized.

3
Using the OpenVMS Debugger with BASIC

This chapter discusses OpenVMS Debugger information that is specific to the BASIC language. For more information about the OpenVMS Debugger, see the *HP OpenVMS Debugger Manual*. Online help is available during debugging sessions.

3.1 Overview of the Debugger

A **debugger** is a tool to help you locate run-time errors quickly. It is used with a program that has already been compiled and linked successfully, with no errors reported, but that does not run correctly. For example, the output might be obviously wrong, the program goes into an infinite loop, or the program terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively, step by step, until you locate the point at which the program stopped working correctly.

The OpenVMS Debugger is a **symbolic debugger**, which means that you can refer to program locations by the symbols (names) you used for those locations in your program—the names of variables, routines, labels, and so on. You do not have to use virtual addresses to refer to memory locations.

The debugger recognizes the syntax, expressions, data typing, and other constructs of BASIC.

3.2 Compiling and Linking to Prepare for Debugging

The following example shows how to compile and link a BASIC program (consisting of a single compilation unit named INVENTORY) so that subsequently you will be able to use the debugger:

```
$ BASIC/DEBUG INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier with the BASIC command instructs the compiler to write the debug symbol records associated with INVENTORY into the object module, INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY in debugger commands. (If your program has several compilation units, you must compile each unit that you want to debug with the /DEBUG qualifier.)

The /DEBUG qualifier with the LINK command instructs the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. The qualifier also causes the OpenVMS image activator to start the debugger at run time. (If your program has several object modules, you might need to specify other modules in the LINK command.)

3.3 Viewing Your Source Code

The debugger provides two methods for viewing source code: noscreen mode and screen mode. By default when you invoke the debugger, you are in noscreen mode, but you might find that it is easier to view your source code with screen mode. Both modes are described in the following sections.

3.3.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. To get into noscreen mode from screen mode, enter SET MODE NOSCREEN. See the sample debugging session in Section 3.7 for a demonstration of noscreen mode.

In noscreen mode, you can use the TYPE command to display one or more source lines. For example, the following command displays line 3 of the module that is currently executing:

```
DBG> TYPE 3
3:    EXTERNAL SUB TRIPLE    &
DBG>
```

The display of source lines is independent of program execution. You can use the TYPE command to display source code from a module other than the one currently executing. In that case, you need to use a directory specification to specify the module. For example, the following command displays lines 16 to 21 of module TEST:

```
DBG> TYPE TEST\16:21
```

3.3.2 Screen Mode

To invoke screen mode, press PF3. In screen mode, by default the debugger splits the screen into three displays called SRC, OUT, and PROMPT.

```
--SRC: module SAMPLE$MAIN -scroll-source--------------------------
     1:  10     !SAMPLE
     2:
     3:         EXTERNAL SUB TRIPLE        &
     4:                     ,PRINT_SUB
     5:
     6:         WHEN ERROR USE HANDLER_1
  -> 7:           CALL TRIPLE
     8:           CALL PRINT_SUB
     9:
- OUT -output-----------------------------------------------
stepped to SAMPLE$MAIN\%LINE 7


- PROMPT -error-program-prompt----------------------------
DBG> STEP
DBG>
```

The SRC display, at the top of the screen, shows the source code of the module (compilation unit) that is currently executing. An arrow in the left column points to the next line to be executed, which corresponds to the current location of the program counter (PC). The line numbers, which are assigned by the compiler, match those in a listing file.

Note

BASIC line numbers are treated as text by the debugger. In this chapter, line numbers refer to the sequential line numbers generated by the compiler. When a program includes or appends code from another file, the included lines of code are also numbered in sequence by the compiler. These line numbers are on the extreme left of a listing file. An explanation of the listing file format is in Chapter 2.

The PROMPT display, at the bottom of the screen, shows the debugger prompt (DBG>), your input, debugger diagnostic messages, and program output. In the example, the debugger commands that have been issued are shown.

The OUT display, in the center of the screen, captures the debugger's output in response to the commands that you issue.

The SRC and OUT displays are scrollable so that you can see whatever information scrolls beyond the display window's edge. Press KP8 to scroll up and KP2 to scroll down. Press KP3 to change the display to be scrolled (by default, the SRC display is scrolled). Scrolling a display does not affect program execution.

If the debugger cannot locate source lines for the currently executing module, it tries to display source lines in the next module down on the call stack for which source lines are available and issues the following message:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
        Displaying source in a caller of the current routine.
```

Source lines might not be available for the following reasons:

- The PC is within a system routine, or a shareable image routine for which no source code is available.
- The PC is within a routine that was compiled without the /DEBUG compiler command qualifier (or with /NODEBUG).
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules). Use the SET SOURCE command to direct the debugger to the new location.

3.4 Controlling and Monitoring Program Execution

This section discusses the following:

- Starting and resuming program execution with the GO command
- Stepping through the program's code with the STEP command
- Determining the current location of the program counter (PC) with the SHOW CALLS command
- Suspending program execution with breakpoints
- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

3.4.1 Starting and Resuming Program Execution

There are two commands for starting or resuming program execution: GO and STEP. The GO command starts execution. The STEP command lets you execute a specified number of source lines or instructions.

GO Command

The GO command starts program execution, which continues until forced to stop. You will probably use the GO command most often in conjunction with breakpoints, tracepoints, and watchpoints. If you set a breakpoint in the path of execution and then enter the GO command (or press the keypad comma key that executes the GO command), execution will be suspended when the program reaches that breakpoint. If you set a tracepoint, the path of execution through that tracepoint will be monitored. If you set a watchpoint, execution will be suspended when the value of the watched variable changes.

You can also use the GO command to test for an exception condition or an infinite loop. If an exception condition that is not handled by your program occurs, the debugger will take over and display the DBG> prompt so that you can issue commands. If you are using screen mode, the pointer in the source display will indicate where execution stopped. You can then use the SHOW CALLS command (see Section 3.4.2) to identify the currently active routine calls (the call stack).

In the case of an infinite loop, the program will not terminate, so the debugger prompt will not reappear. To obtain the prompt, interrupt the program by pressing Ctrl/Y and then issue the DCL command DEBUG. You can then look at the source display and a SHOW CALLS display to locate the PC.

STEP Command

The STEP command (which you can use either by entering STEP or by pressing KP0) allows you to execute a specified number of source lines or instructions, or to execute the program to the next instruction of a particular kind, for example, to the next CALL instruction.

By default, the STEP command executes a single source line at a time. In the following example, the STEP command executes one line, reports the action ("stepped to . . . "), and displays the line number (27) and source code of the next line to be executed:

```
DBG> STEP
stepped to TEST\COUNTER\%LINE 27
     27:   X = X + 1
DBG>
```

The PC is now at the first machine code instruction for line 27 of the module TEST; line 27 is in COUNTER, a routine within the module TEST. TEST\COUNTER\%LINE 27 is a directory specification. The debugger uses directory specifications to refer to symbols. (However, you do not need to use a path name in referring to a symbol, unless the symbol is not unique; in that case, the debugger will issue an error message.) See the *HP OpenVMS*

Debugger Manual or online help for more information about resolving multiply-defined symbols.

You can specify a number of lines for the STEP command to execute. In the following example, the STEP command executes three lines:

```
DBG> STEP 3
```

Note that only those source lines for which code instructions were generated by the compiler are recognized as executable lines by the debugger. The debugger skips over any other lines—for example, comment lines.

Also, if a line has more than one statement on it, the debugger will execute all the statements on that line as part of the single step.

Using the STEP/OVER command to step over a GOSUB statement will still proceed to the target of the GOSUB since this statement is just a special kind of GOTO statement and not a routine call.

You can specify different stepping modes, such as stepping by instruction rather than by line (SET STEP INSTRUCTION). To resume to the default behavior, enter the SET STEP LINE command. Also by default, the debugger steps over called routines—execution is not suspended within a called routine, although the routine is executed. By entering the SET STEP INTO command, you tell the debugger to suspend execution within called routines as well as within the currently executing module. To resume the default behavior, enter the SET STEP OVER command.

3.4.2 Determining the Current Location of the Program Counter

The SHOW CALLS command lets you determine the current location of the program counter (PC) (for example, after returning to the debugger following a Ctrl/Y interrupt). The command shows a traceback that lists the sequence of calls leading to the currently executing routine. For example:

```
DBG> SHOW CALLS
 module name      routine name       line     rel PC    abs PC

*TEST             PRODUCT              18    00000009  0000063C
*TEST             COUNTER              47    00000009  00000647
*MY_PROG          MY_PROG              21    0000000D  00000653
DBG>
```

For each routine (beginning with the currently executing routine), the debugger displays the following information:

- Name of the module that contains the routine
- Name of the routine

- Line number at which the call was made (or at which execution is suspended, in the case of the current routine)
- Corresponding PC addresses (the relative PC address from the start of the routine and the absolute PC address of the program)

This example indicates that execution is currently at line 18 of routine PRODUCT (in module TEST), which was called from line 47 of routine COUNTER (in module TEST), which was called from line 21 of routine MY_PROG (in module MY_PROG).

3.4.3 Suspending Program Execution

The SET BREAK command lets you select **breakpoints**, which are locations at which the program will stop running. When you reach a breakpoint, you can enter commands to check the call stack, examine the current values of variables, and so on.

A typical use of the SET BREAK command is shown in the following example:

```
DBG> SET BREAK COUNTER
DBG> GO
     .
     .
     .
break at TEST\COUNTER
     34:  SUB COUNTER(LONG X,Y)
DBG>
```

In this example, the SET BREAK command sets a breakpoint on the subprogram COUNTER; the GO command starts execution. When the subprogram COUNTER is encountered, execution is suspended, the debugger announces that the breakpoint at COUNTER has been reached (break at . . .), displays the source line (34) where execution is suspended, and prompts you for another command. At this breakpoint, you can step through the subprogram COUNTER, using the STEP command, and use the EXAMINE command (see Section 3.5.1) to check on the current values of X and Y.

When using the SET BREAK command, you can specify program locations using various kinds of address expressions (for example, line numbers, routine names, instructions, virtual memory addresses). With high-level languages, you typically use routine names, labels, or line numbers, possibly with directory specifications to ensure uniqueness.

Routine names and labels should be specified as they appear in the source code. Line numbers may be derived from either a source code display or a listing file. When specifying a line number, use the prefix %LINE. (Otherwise, the debugger will interpret the line number as a memory location.) For example,

the next command sets a breakpoint at line 41 of the currently executing module; the debugger will suspend execution when the PC is at the start of line 41:

```
DBG> SET BREAK %LINE 41
```

Note that you can set breakpoints only on lines that resulted in machine code instructions. The debugger warns you if you try to do otherwise (for example, on a comment line). If you want to pick a line number in a module other than the one currently executing, you need to specify the module's name in a directory specification. For example:

```
DBG> SET BREAK SCREEN_IO\%LINE 58
```

You do not always have to specify a particular program location, such as line 58 or COUNTER, to set a breakpoint. You can set breakpoints on events, such as exceptions. You can use the SET BREAK command with a qualifier, but no parameter, to break on every line, or on every CALL instruction, and so on. For example:

```
DBG> SET BREAK/LINE
DBG> SET BREAK/CALL
```

You can conditionalize a breakpoint (with a WHEN clause) or specify that a list of commands be executed at the breakpoint (with a DO clause on the debugger command). For example, the next command sets a breakpoint on the label LOOP3. The DO (EXAMINE TEMP) clause causes the value of the variable TEMP to be displayed whenever the breakpoint is triggered.

```
DBG> SET BREAK LOOP3 DO (EXAMINE TEMP)
DBG> GO
    .
    .
    .
break at COUNTER\LOOP3
     37:    LOOP3: FOR I = 1 TO 10
COUNTER\TEMP:   284.19
DBG>
```

To display the currently active breakpoints, enter the SHOW BREAK command:

```
DBG> SHOW BREAK
breakpoint at SCREEN_IO\%LINE 58
breakpoint at COUNTER\LOOP3
   do (EXAMINE TEMP)
    .
    .
    .
DBG>
```

To cancel a breakpoint, enter the CANCEL BREAK command, specifying the program location exactly as you did when setting the breakpoint. The CANCEL BREAK/ALL command cancels all breakpoints.

3.4.4 Tracing Program Execution

The SET TRACE command lets you select **tracepoints**, which are locations for tracing the execution of your program without stopping its execution. After setting a tracepoint, you can start execution with the GO command and then monitor the PC's path, checking for unexpected behavior. By setting a tracepoint on a routine, you can also monitor the number of times the routine is called.

As with breakpoints, every time a tracepoint is reached, the debugger issues a message and displays the source line. It can also display other information that you have specified (as shown in the last example in this section, in which the value of a specified variable is displayed). However, at tracepoints, unlike breakpoints, the program continues executing, and the debugger prompt is not displayed. For example:

```
DBG> SET TRACE COUNTER
DBG> GO
     .
     .
     .
trace at TEST\COUNTER
     34:  SUB COUNTER(LONG X,Y)
     .
     .
     .
```

When using the SET TRACE command, you specify address expressions, qualifiers, and optional clauses exactly as with the SET BREAK command.

The /LINE qualifier instructs the SET TRACE command to trace every line and is a convenient means of checking the execution path. By default, lines are traced within all called routines as well as the currently executing routine. If you do not want to trace system routines or routines in shareable images, use the /NOSYSTEM or /NOSHARE qualifiers. For example:

```
DBG> SET TRACE/LINE/NOSYSTEM/NOSHARE
```

The /SILENT qualifier suppresses the trace message and source code display. This is useful when you want to use the SET TRACE command to execute a debugger command at the tracepoint. For example:

```
DBG> SET TRACE\SILENT %LINE 83 DO (EXAMINE STATUS)
DBG> GO
     .
     .
     .
SCREEN_IO\CLEAR\STATUS:   'OFF'
     .
     .
     .
```

3.4.5 Monitoring Changes in Variables

The SET WATCH command lets you set watchpoints that will be monitored continuously as your program executes.

If the program modifies the value of a watched variable, the debugger suspends execution and displays the old and new values.

```
DBG> SET WATCH TOTAL
```

Subsequently, every time the program modifies the value of TOTAL, the watchpoint is triggered. The debugger monitors watchpoints continuously during program execution.

The next example shows what happens when your program modifies the contents of a watched variable:

```
DBG> SET WATCH TOTAL
DBG> GO
     .
     .
     .
watch of SCREEN_IO\TOTAL\%LINE 13
     13:   TOTAL = TOTAL + 1
    old value: 16
    new value: 17
break at SCREEN_IO.%LINE 14
     14:   CALL Pop_rtn(TOTAL)
DBG>
```

In this example, a watchpoint is set on the variable TOTAL and the GO command starts execution. When the value of TOTAL changes, execution is suspended. The debugger announces the event (watch of . . .), identifying where TOTAL changed (line 13) and the associated source line. The debugger then displays the old and new values and announces that execution has been suspended at the start of the next line (14). (The debugger reports break

at . . . , but this is not a breakpoint; it is still the effect of the watchpoint.) Finally, the debugger prompts for another command.

When a change in a variable occurs at a point other than the start of a source line, the debugger gives the line number plus the byte offset from the start of the line.

3.5 Examining and Manipulating Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables, and evaluate expressions in BASIC programs.

3.5.1 Displaying the Values of Variables

To display the current value of a variable, use the EXAMINE command as follows:

```
DBG> EXAMINE variable_name
```

The debugger recognizes the compiler-generated data type of the specified variable and retrieves and formats the data accordingly. The following examples show some uses of the EXAMINE command:

Examine a string variable:

```
DBG> EXAMINE EMPLOYEE_NAME
PAYROLL\EMPLOYEE_NAME:    "Peter C. Lombardi"
DBG>
```

Examine three integer variables:

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:   4
SIZE\LENGTH:  7
SIZE\AREA:   28
DBG>
```

Examine a two-dimensional array of integers (two rows and three columns):

```
DBG> EXAMINE INTEGER_ARRAY
PROG2\INTEGER_ARRAY
   (0,0):       27
   (0,1):       31
   (0,2):       12
   (1,0):       15
   (1,1):       22
   (1,2):       18
DBG>
```

Examine element 4 of a one-dimensional string array:

```
DBG> EXAMINE CHAR_ARRAY(4)
PROG2\CHAR_ARRAY(4): 'm'
DBG>
```

Note that the EXAMINE command can be used with any kind of address expression (not just a variable name) to display the contents of a program location. The debugger associates certain default data types with untyped locations. You can override the defaults for typed and untyped locations if you want the data to be interpreted and displayed in some other data format. The debugger supports the data types and operators of BASIC including RECORDs and RFAs.

See Section 3.5.3 for an explanation of how the EXAMINE and the EVALUATE commands differ.

3.5.2 Changing the Values of Variables

To change the value of a variable, use the DEPOSIT command as follows:

```
DBG> DEPOSIT variable_name = value
```

The DEPOSIT command is like an assignment statement in BASIC.

In the following examples, the DEPOSIT command assigns new values to different variables. The debugger checks that the value assigned, which may be a language expression, is consistent with the data type and dimensional constraints of the variable.

Deposit a string value (it must be enclosed in quotation marks or apostrophes):

```
DBG> DEPOSIT PARTNUMBER = "WG-7619.3-84"
```

Deposit an integer expression:

```
DBG> DEPOSIT WIDTH = CURRENT_WIDTH + 10
```

Deposit element 12 of an array of characters (you cannot deposit an entire array aggregate with a single DEPOSIT command, only an element):

```
DBG> DEPOSIT C_ARRAY(12) = 'K'
```

You can specify any kind of address expression, not just a variable name, with the DEPOSIT command (as with the EXAMINE command). You can override the defaults for typed and untyped locations if you want the data to be interpreted in some other data format.

3.5.3 Evaluating Expressions

To evaluate a language expression, use the EVALUATE command as follows:

```
DBG> EVALUATE lang_exp
```

The debugger recognizes the operators and expression syntax of the currently set language. In the following example, the value 45 is assigned to the integer variable WIDTH; the EVALUATE command then obtains the sum of the current value of WIDTH plus 7:

```
DBG> DEPOSIT WIDTH = 45
DBG> EVALUATE WIDTH + 7
52
DBG>
```

Following is an example of how the EVALUATE and the EXAMINE commands are similar. When the expression following the command is a variable name, the value reported by the debugger is the same for either command.

```
DBG> DEPOSIT WIDTH = 45
DBG> EVALUATE WIDTH
45
DBG> EXAMINE WIDTH
SIZE\WIDTH:   45
```

Following is an example of how the EVALUATE and EXAMINE commands are different:

```
DBG> EVALUATE WIDTH + 7
52
DBG> EXAMINE WIDTH + 7
SIZE\WIDTH:   131584
```

With the EVALUATE command, WIDTH + 7 is interpreted as a language expression, which evaluates to 45 + 7, or 52. With the EXAMINE command, WIDTH + 7 is interpreted as an address expression: 7 bytes are added to the address of WIDTH, and whatever value is in the resulting address is reported (in this example, 131584).

3.6 Stepping Into BASIC Routines

This section provides details of the STEP/INTO command that are specific to BASIC.

In the following example, the debugger is waiting to proceed at source line 63. If you enter a STEP command at this point, the debugger will proceed to source line 64 without stopping during the execution of the function call. To step through the source code in the DEF function *deffun*, you must use the STEP/INTO command. A STEP/INTO command entered while the debugger

has stopped at source line 63 causes the debugger to display the source code for *deffun* and stop execution at source code line 3.

```
  1     DECLARE LONG FUNCTION deffun (LONG)
  2     DECLARE LONG A
  3     DEF LONG deffun (LONG x)
  4         deffun = x
  5     END DEF
   .
   .
   .
->63    A = deffun (6%)
  64    Print "The value of A is: "; A
```

The STEP/INTO command is useful for stepping into external functions and DEF functions in HP BASIC. If you use this command to step into GOSUB blocks, the debugger steps into Run-Time Library (RTL) routines, providing you with no useful information.

In the following program, the debugger has suspended execution at source line 8. If you now enter a STEP/INTO command, the debugger steps into the relevant RTL code and informs you that no source lines are available.

```
  1  10   RANDOMIZE
   .
   .
   .
->8       GOSUB Print_routine
  9       STOP
   .
   .
   .
  20     Print_routine:
  21         IF Competition = Done
  22           THEN PRINT "The winning ticket is #";Winning_ticket
  23           ELSE PRINT "The game goes on."
  24         END IF
  25     RETURN
```

As in the previous example, a STEP command alone will cause the debugger to proceed directly to source line 9.

Table 3–1 summarizes the resultant behavior of the STEP/INTO command when used to step into external functions, DEF functions, and GOSUB blocks.

Table 3–1 Resultant Behavior of the STEP/INTO Command

Action	Results
STEP/INTO DEF function	Steps into function
STEP/INTO DEF* function	Steps into RTL
STEP/INTO external function or SUB routine[1]	Steps into function
STEP/INTO GOSUB block	Steps into RTL

[1]Unless the subroutine is compiled with the /NOSETUP qualifier or equivalent, it will appear to step into RTL code, because an environment setup RTL routine is normally called as the very first thing of the subroutine.

3.6.1 Controlling Symbol References

When using the OpenVMS Debugger, all HP BASIC variable and label names within a single program unit must be unique; otherwise, the debugger will be unable to determine the symbol to which you are referring.

3.7 Sample Debugging Session

This section shows a sample debugging session using a BASIC program that contains a logic error.

The following program compiles and links without diagnostic messages from either the compiler or the linker. However, after printing the headers, the program is caught in a loop printing the same figures indefinitely.

```
 1   10 !SAMPLE program for DEBUG illustration
 2      DECLARE INTEGER Number
 3      Print_headers:
 4      PRINT "NUMBER", "SQUARE", "SQUARE ROOT"
 5      PRINT
 6      Print_loop:
 7      FOR Number = 10 TO 1 STEP -1
 8        PRINT Number, Number^2, SQR(Number)
 9        Number = Number + 1
10      NEXT Number
11      PRINT
12      END
```

The following text shows the terminal dialogue for a debugging session, which helps locate the error in the program SAMPLE. The callouts are keyed to explanatory notes that follow the dialogue.

```
$ BASIC/LIST/DEBUG SAMPLE ❶
$ LINK/DEBUG SAMPLE ❷
$ RUN SAMPLE

              VAX DEBUG Version n.n

%DEBUG-I-INITIAL, language is BASIC module set to 'SAMPLE$MAIN' ❸
DBG>STEP 2 ❹
NUMBER         SQUARE       SQUARE ROOT
stepped to SAMPLE$MAIN\%line 7
     7:         FOR Number = 10 TO 1 STEP -1 ❺
DBG> STEP 4 ❻
10        100          3.16228
stepped to  SAMPLE$MAIN\%LINE 7
     7:         FOR Number = 10 TO 1 STEP -1
DBG> EXAMINE Number ❼
SAMPLE$MAIN\NUMBER:     10 ❽
DBG> STEP 4 ❾
10        100          3.16228
stepped to  SAMPLE$MAIN\%LINE 7
     7:         FOR Number = 10 TO 1 STEP -1
DBG> EXAMINE Number ❿
SAMPLE$MAIN\NUMBER:     10 ⓫
DBG> DEPOSIT Number = 9 ⓬
DBG> STEP 4 ⓭
9          81          3
stepped to  SAMPLE$MAIN\%LINE 7
     7:         FOR Number = 10 TO 1 STEP -1
DBG> EXAMINE Number ⓮
SAMPLE$MAIN\NUMBER:     9 ⓯
DBG> STEP ⓰
9          81          3
stepped to  SAMPLE$MAIN\%LINE 8
     8:           PRINT Number, Number^2, SQR(Number) ⓱
DBG> STEP ⓲
stepped to SAMPLE$MAIN\%LINE 9
     9:         Number = Number + 1 ⓳
DBG> EXIT ⓴
```

The following explains the terminal dialogue in the above example:

❶ Compile SAMPLE.BAS with the /LIST and /DEBUG qualifiers. The listing file can be useful while you are in the debugging session.

❷ Link SAMPLE.BAS with the /DEBUG qualifier.

❸ The debugger identifies itself and displays the debugger prompt after you invoke the debugger with the RUN command.

❹ Step through 2 executable statements to the FOR statement.

❺ The headers print successfully and the program reaches the FOR statement.

❻ Step through one iteration of the loop.

❼ Request the contents of the variable *Number*.

❽ The debugger shows the contents of the loop index to be 10.

❾ Step through another iteration of the loop.

❿ Examine the value of the loop index again.

⓫ The debugger shows that the loop index is still 10. The loop index has not changed from its initial setting in the FOR statement.

⓬ Deposit the correct value into *Number*.

⓭ Step through another iteration of the loop.

⓮ Examine the contents of *Number* again.

⓯ Observe that the number has not been changed yet.

⓰ Step through just one statement to discover what is interfering with the value of *Number* during execution of the loop.

⓱ Observe that this statement does not affect the value of *Number*.

⓲ Step through another statement in the loop.

⓳ Observe that this statement counteracts the change in the loop index.

⓴ Exit from the debugger. You can now edit the program to delete line 9 and reprocess the program. Alternatively, you could use the EDIT command while in the debugger environment.

This debugging session shows that the FOR...NEXT loop index (*Number*) is not being changed correctly. An examination of the statements in the loop shows that the variable *Number* is being decreased by one during each execution of the FOR statement, but incremented by one with each execution of the loop statements. From this you can determine that the loop index will not change at all and the program will loop indefinitely. To correct the problem, you must delete the incorrect statement and recompile the source program.

3.8 Hints for Using the OpenVMS Debugger

A STEP at a statement that causes an exception might never return control to the debugger. The debugger cannot determine what statement in the BASIC source code will execute after the exception occurs. Therefore, set explicit breaks if STEP is used on statements that cause exceptions.

The following hints should help when you use the STEP command to debug programs that handle errors:

- When you STEP at a statement that takes an error, the debugger will not regain control unless the program reaches an explicit breakpoint or the next statement that would have executed if no error had occurred. Set explicit breaks if you want the program to stop in any other place.
- Use of the STEP command at a statement that takes an error does not return control to the debugger when the program reaches the error handler code. If you want the program to break when program execution enters an error handler, explicitly set a breakpoint at the error handler. This applies to both ON ERROR handlers and WHEN handlers.
- If you are within a WHEN handler, a STEP at a statement that terminates execution within the WHEN handler (CONTINUE, RETRY, END WHEN, END HANDLER, EXIT HANDLER) will not stop unless program flow reaches a point where an explicit breakpoint is set.
- STEP at a RESUME statement in an ON ERROR handler results in the program execution stopping at the first line of non-error-handler code.
- Use SET BREAK/EXCEPTION at the beginning of the debugging session to prevent unexpected errors from occurring. This breakpoint is not necessary if you have set explicit breakpoints at all error handlers. However, use of this command will break at all exceptions, allowing you to check that you have the proper breakpoints to stop program execution following the exception.

Part II

Compaq BASIC Programming Concepts

Part II explains Compaq BASIC programming concepts including input and output, arrays, data definition, program control, and functions.

4

BASIC Concepts and Elements

A BASIC **program** is a series of instructions for the compiler. These instructions are built using the fundamental elements of BASIC. This chapter describes these elements or building blocks.

4.1 Line Numbers

BASIC gives you the option of developing programs with line numbers or without line numbers.

4.1.1 Programs with Line Numbers

If you use line numbers in your program, you must follow these rules:

- A line number must be a unique integer from 1 to 32767. HP BASIC does not allow programs to have duplicate line numbers.
- A line number can contain leading zeros; however, embedded spaces, tabs, and commas are invalid in line numbers.
- There must be a line number on the first line of the program.
- If a source file contains subprograms, then each subprogram must begin on a numbered line.

In a multiple-unit program with line numbers, any comments following an END, END SUB, or END FUNCTION statement become a part of the previous subprogram during compilation unless they begin on a numbered line. This is not the case in multiple-unit programs without line numbers.

Although line numbers are not required, you might want to use them on every line that can cause a run-time error, depending on the type of error handling you use. See Chapter 15 for more information about handling run-time errors.

4.1.2 Programs Without Line Numbers

If you do not use line numbers in your program, follow these rules:

- Use a text editor to enter and edit the program.
- No line numbers are allowed anywhere in the program module.
- The ERL function is not allowed.
- REM statements are not allowed.

In a multiple-unit program without line numbers, any comments following an END, END SUB, or END FUNCTION statement become a part of the next subprogram during compilation (unless there is no next subprogram). This is not the case in multiple-unit programs with line numbers.

You can avoid all of these restrictions by placing a line number on the first line of your program; no additional line numbers are required. The line number on the first program line causes the compiler to compile your program as a program with line numbers.

When you write a program with or without line numbers, you can begin your program statements in the first character position on a line.

To develop the following program, use a text editor, and observe the restrictions previously listed:

```
!This is a short program that does not contain any
!BASIC line numbers.
!This program must be entered using a text editor;
!it cannot be entered directly into the environment.
!
PRINT "This program converts kilogram weight to pounds"
INPUT "How many kilograms";A
!This is the conversion factor
B = A / 2.2
PRINT "For ";A;" kilograms, the pound weight is ";B
END
```

Output

```
This program converts kilogram weight to pounds
How many kilograms? 11
For  11  kilograms, the pound weight is  5
```

You can use exclamation comment fields instead of REM statements to insert comments into programs without line numbers. An exclamation point in column 1 causes the HP BASIC compiler to ignore the rest of the line. You can also identify program statements in programs without line numbers by using labels.

4.1.3 Labels

A **label** is a 1- to 31-character identifier that you use to identify a block of statements. All label names must begin with a letter; the remaining characters, if any, can be any combination of letters, digits, dollar signs ($), underscores (_), or periods (.), but the final character cannot be a dollar sign.

Labels have the following advantages over line numbers:

- Meaningful label names provide documentation.
- You can use labels in programs with or without line numbers.

When you use a label to mark a program location, you must end the label with a colon (:). The colon is used to show that the label name is being defined instead of referenced. When you reference the label, do not include the colon.

In the following example, the label names end with colons when they mark a location, but the colons are not present when the labels are referenced:

```
OPTION TYPE = EXPLICIT        ! Require declarations
DECLARE INTEGER A
   .
   .
   .
Outer_loop:
      IF A <> B
      THEN
Inner_loop:
      IF B = C
        THEN
          A = A + 1
          GOTO Outer_loop
        ELSE
          B = B + 1
          GOTO Inner_loop
        END IF
      END IF
```

Labels have no effect on the order in which program lines are executed; they are used to identify a statement or block of statements.

4.1.4 Continuation of Long Program Statements

If a program line is too long for one line of text, you can continue the program line by placing an ampersand (&) at the end of the line. Note that only spaces and tabs are valid between the ampersand and the carriage return.

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

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

In an IF...THEN...ELSE construction, ampersands (&) are not necessary. If a continuation line begins with THEN or ELSE, then no ampersand is necessary. Similarly, in a line following a THEN or an ELSE, there is no ampersand.

```
IF (A$ = B$)
THEN
   PRINT "The two values are equal"
ELSE
   PRINT "The two values are different"
END IF
```

Several statements can be associated with a single program line. If there are several statements on one line, they must be separated by backslashes (\ ). For example:

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

Because all statements are on the same program line, any reference to this program line refers to all three statements.

4.2 Identifying Program Units

You can delimit a main program compilation unit with the PROGRAM and END PROGRAM statements. This allows you to identify a program with a name other than the file name. The program name must not duplicate the name of a SUB, FUNCTION, or PICTURE subprogram. For example:

```
PROGRAM Sort_out
   .
   .
   .
END PROGRAM
```

If you include the PROGRAM statement in your program, the name you specify becomes the module name of the compiled source. This feature is useful when you use object libraries because the librarian stores modules by their module name rather than the file name. Similarly, module names are used by the OpenVMS Debugger and the OpenVMS Linker.

For more information about PROGRAM units, see Chapter 12.

4.3 BASIC Character Set

BASIC uses the full ASCII character set, which includes the following:

- The letters A to Z, both uppercase and lowercase
- The digits 0 to 9
- Special characters

See the *HP BASIC for OpenVMS Reference Manual* for a complete list of the ASCII character set and character values.

The compiler does not distinguish between uppercase and lowercase letters, except for letters inside quotation marks (called **string literals**) or letters in a DATA statement. The compiler also does not process characters in a REM statement or comment field.

You can use nonprinting characters in your program—for example, in string literals and constants—but to do so you must do one of the following:

- Use a predefined constant such as ESC or DEL
- Use the CHR$ function to specify an ASCII value

See Section 4.6 for more information about predefined constants. See Chapter 10 for more information about the CHR$ function.

4.4 Program Documentation

Documenting a program is the process of putting explanatory text (comments) into your code to make the program more understandable. Program documentation does not affect the way a program executes. You can add comments throughout a program; however, programs that are neatly structured need fewer comments. You can clarify your code by doing the following:

- Using meaningful variable names
- Including sufficient white space
- Indenting your program lines according to the structure of your code

A comment field starts with an exclamation point (!) and ends with another exclamation point or a carriage return. The following example contains both comments and program statements. Any text that follows an exclamation point is ignored.

```
PROGRAM sample
!+
!   Require that all variables be declared
!-
OPTION TYPE = EXPLICIT
!+
!   Set up error handler
!-
WHEN ERROR USE Error_routine
!+
!   Declarations
!-
   .
   .
   .
END PROGRAM
```

You can also mix comments and code on the same line. For example:

```
DECLARE                                         &
    INTEGER                                     &
    Print_page,         ! Current page number   &
    Print_line,         ! Current line number   &
    Print_column        ! Current column number
```

All text between the exclamation point and the carriage return is ignored, with one exception: the ampersand is still recognized. This is a continuation character that specifies that a single statement is being continued on the next line. Only spaces and tabs are valid between the ampersand and the carriage return.

Note

Although you can also terminate a comment field with an exclamation point, this practice is not recommended. Any text that follows the second exclamation point is treated as part of your program code.

4.5 Declarations and Data Types

Following are methods for creating variables and specifying data types:

- Implicit data typing
- Explicit data typing

With implicit data typing, BASIC creates and specifies a data type for a variable the first time you reference it in your program. With explicit data typing, you must use one of four declarative statements (see Section 4.5.2) to name and type your program values.

Following are the data types you can specify:

- Integer (INTEGER)
- Floating-point (REAL)
- String (STRING)
- Packed Decimal (DECIMAL)
- Record File Address (RFA)

Within the INTEGER and REAL data types there are further subdivisions: BYTE, WORD, LONG, or QUAD for INTEGER and SINGLE, DOUBLE, GFLOAT, SFLOAT, TFLOAT, or XFLOAT for REAL. Choosing one of these subtypes lets you control the following:

- The amount of storage required for the value; its container size
- The range and precision that the value can accept

For more information about data types, see Chapter 8.

4.5.1 Implicit Data Typing

With implicit data typing, a data type for a variable is created and specified the first time you reference it. You specify the data type of the variable by a suffix on the variable name as follows:

- A percent sign suffix (%) specifies the INTEGER data type.
- A dollar sign suffix ($) specifies the STRING data type.
- Any other ending character specifies a variable of the default data type.

The default data type is SINGLE on Alpha BASIC and SFLOAT on I64 BASIC. However, you can specify your own default at DCL command level or with the OPTION statement in your program. For more information about establishing default data types, see Chapter 2, as well as the OPTION statement in the *HP BASIC for OpenVMS Reference Manual*.

The first time the variable is referenced, it creates a variable with that name and data type and allocates storage for that variable.

In the following example, two INTEGER variables are created, *A%* and *B%*. Even though the values assigned to these variables are REAL, the values are converted to INTEGER to match the data type specified for the variables. The sum of these two values is therefore 30, not 30.6, as it would be if the variables were named *A* and *B*.

```
A% = 10.1
B% = 20.5
PRINT A% + B%

 30
```

4.5.2 Explicit Data Typing

With explicit data typing, you use a declarative statement to name and specify a data type for your program values.

BASIC provides the following declarative statements. These statements create variables and allocate storage:

DECLARE
DIMENSION
COMMON
MAP

The statement you choose depends on the way in which you will use the variables:

- DECLARE and DIMENSION allocate dynamic storage for variables; storage is allocated when the program executes.
- COMMON and MAP statements allocate storage for variables statically; storage is allocated when the program is compiled.

All declarative statements associate a data type with a variable. For more information, see Chapter 7.

4.6 Constants

A **constant** is a value that does not change during program execution. Constants can be either literals or named constants and can be of any data type except RFA. You can use the DECLARE CONSTANT statement to create named constants. Constants can be of the following types:

- Integer
- Floating-point
- Packed decimal

- String

In addition, predefined constants are provided and are useful for the following:

- Formatting program output to improve clarity
- Making source code easier to understand
- Using nonprinting characters without having to look up their ASCII values

Table 4–1 lists the predefined constants.

Table 4–1 Predefined Constants

Constant	Decimal ASCII Value	Description
BEL (Bell)	7	Sounds the terminal bell
BS (Backspace)	8	Moves cursor one position to the left
HT (Horizontal Tab)	9	Moves cursor to the next horizontal tab stop
LF (Line Feed)	10	Moves cursor to the next line
VT (Vertical Tab)	11	Moves cursor to the next vertical tab stop
FF (Form Feed)	12	Moves cursor to the start of the next page
CR (Carriage Return)	13	Moves cursor to the beginning of the current line
SO (Shift Out)	14	Shifts out for communications networking, screen formatting, and alternate graphics
SI (Shift In)	15	Shifts in for communications networking, screen formatting, and alternate graphics
ESC (Escape)	27	Marks the beginning of an escape sequence
SP (Space)	32	Inserts one blank space in program output
DEL (Delete)	127	Deletes the last character entered
PI	None	Represents the number PI with the precision of the default floating-point data type

These predefined constants simplify the task of using nonprinting characters in your programs. For example, the following statement causes a bell to sound on your terminal:

```
PRINT BEL
```

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

For more information about constants, see Chapter 7 and the *HP BASIC for OpenVMS Reference Manual*.

4.7 Variables

A **variable** is a storage location that is referred to by a variable name. Variable values can change during program execution. Each named location can hold only one value at a time.

A variable name can have up to 31 characters. The name must begin with a letter; the remaining characters, if any, can be any combination of letters, digits, dollar signs ($), underscores (_), and periods (.).

Variables can be grouped in an orderly series (such as a list or table) under a single name, called an **array**. You refer to a single variable in an array by using one or more **subscripts** that specify the variable's position in the array. (See Section 4.7.5 for more information on arrays.)

4.7.1 Floating-Point Variables

A **floating-point variable** is a named location that stores a floating-point value. The storage space required to hold the value depends on the variable's REAL subtype. For example, each SINGLE floating-point variable requires 32 bits (4 bytes) of storage, while each DOUBLE floating-point variable requires 64 bits (8 bytes) of storage.

Note that if any integer value is assigned to a floating-point variable, the value is converted to a floating-point number.

4.7.2 Integer Variables

An **integer variable** is a named location that stores a whole number. The storage space required to hold the value depends on the variable's INTEGER subtype. For example, each BYTE integer variable requires 8 bits (1 byte) of storage, while each LONG integer variable requires 32 bits (4 bytes) of storage.

If you assign a floating-point value to an integer variable, the fractional portion of the value is trunctated; it does not round to the nearest integer. In the following example, the value -5, not -6, is assigned to the integer variable.

```
B% = -5.7
```

Although the integer data types QUAD, LONG, WORD, and BYTE allow the minimum values -9223372036854775808, -2147483648, -32768, and -128, respectively, you cannot use these constants explicitly, because HP BASIC reports an integer overflow error while attempting to parse the literal constant. To use these values, you must use either radix notation, such as –"32768"L, or a constant expression. For example:

```
DECLARE WORD CONSTANT Word_const = -32767 - 1
```

4.7.3 Packed Decimal Variables

A **packed decimal** (DECIMAL data type) variable is made up of several storage locations, the number of which depends on the declared size of the variable. However, a packed decimal variable is still referred to by a single variable name.

When you declare a packed decimal variable, you specify the total number of digits and the number of digits to the right of the decimal place that you want.

The following statement creates a packed decimal variable named *My_decimal*, which can contain up to 8 digits: 6 digits to the left of the decimal point and 2 digits to the right of the decimal point.

```
OPTION TYPE = EXPLICIT

DECLARE DECIMAL (8,2) My_decimal
```

Packed decimal numbers are most useful for dollars-and-cents calculations.

4.7.4 String Variables

Unlike some of the numeric variables described so far, a **string variable** does not correspond to a single location in memory because a string variable is more likely to exceed a single location in memory. Therefore, the value of a string variable can be contained in any number of memory locations. However, a string variable is still referred to by a single name. For example:

```
DECLARE STRING Employee_name
```

4.7.5 Subscripted Variables

A **subscripted variable** is a floating-point, integer, packed decimal, RFA, or string variable that is part of an array. Chapter 6 describes arrays in more detail.

An array is a set of data organized in one or more dimensions. A one-dimensional array is called a **list** or **vector**. A two-dimensional array is called a **matrix**. Arrays can have up to 32 dimensions.

When you create an array, its size is determined by the number of dimensions and the maximum size, called the **bound**, of each dimension. Subscripts begin by default with 0, not 1. That is, when calculating the number of elements in a dimension, you count from zero to the bound specified.

The following DECLARE statement creates an 11 by 11 array of integers. Therefore, the array contains a total of 121 array elements.

```
DECLARE INTEGER My_array (10, 10)
```

There are many applications where you need to reference data for a particular range of values. You can specify a lower bound other than zero for your arrays. The following example declares an array containing the birth rates for the years from 1945 to 1985:

```
OPTION TYPE = EXPLICIT,        &
       SIZE = REAL SINGLE

DECLARE REAL Birth_rates(1945 TO 1985)
```

Subscripts define the position of an element in an array; the expression *Birth_rates*(1970) refers to the 26th value of the array *Birth_rates*. For more information about arrays, see Chapter 6.

Note

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

4.7.6 Initialization of Variables

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

- Numeric variables and array elements (except those in MAP or COMMON statements).
- String variables and array elements (except those in MAP or COMMON statements).
- Variables in subprograms. Subprogram variables (except those in MAP or COMMON statements) are initialized to zero or the null string each time the subprogram is called.
- Arrays created with an executable DIMENSION statement. The array is reinitialized each time the array is redimensioned.

4.8 Keywords and Reserved Words

Keywords are elements of the BASIC language. Keywords that are not reserved can be used as user identifiers such as labels, variable or constant names, or names of MAP or COMMON areas. Depending upon the location of the keyword in your program statement, the compiler will treat it as either a keyword or a user identifier. Your programs use keywords and reserved words to:

- Define data
- Perform operations
- Invoke functions

See the *HP BASIC for OpenVMS Reference Manual* for a list of keywords and reserved words.

Keywords determine whether the statement is executable or nonexecutable. Executable statements such as PRINT, GOTO, and READ perform operations. Nonexecutable statements such as DATA, DECLARE, and REM describe the characteristics and arrangement of data, usage information, and comments.

Every statement except LET must begin with a keyword. A 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.

There are also phrases of keywords. In this case, the spacing requirements vary.

4.9 Operands, Operators, and Expressions

An **operand** contains a value. An operand can be a scalar, subscripted variable, named constant, literal, and so on. An operator specifies a procedure to be carried out by one or more operands. An expression consists of operands separated by operators.

The following are types of operators:

Arithmetic
String
Relational
Logical

When combined with operands, these operators can produce:

- Numeric expressions
- String expressions

- Conditional expressions

For more information about operands, operators, and expressions, see the *HP BASIC for OpenVMS Reference Manual*.

4.10 Assignment Statements

The following statements assign values to variables:

- LET
- INPUT
- LINPUT
- INPUT LINE

LET and INPUT statements allow you to assign values to any type of variable, while LINPUT and INPUT LINE allow you to assign values to string variables. For example:

```
LET A = 1.25
```

LET is an optional keyword. You can assign a value to more than one variable at a time, although this is not recommended. Instead, use a separate assignment statement each time you assign a value to a variable.

Whenever you assign a value to a numeric variable, BASIC converts the value to the data type of the variable. If you assign a floating-point value to an integer variable, BASIC truncates the value at the decimal point. If you assign an integer value to a floating-point variable, BASIC converts the value to floating-point format.

You can also assign values to variables with the DATA and READ statements; however, this method requires that you know all input data values while you are coding your program.

The INPUT, LINPUT, and INPUT LINE statements all assign values in the context of data being read into the program. These statements are discussed in Chapter 5.

5
Simple Input and Output

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

5.1 Program Input

BASIC programs receive data in the following ways:

- You can enter data interactively while the program runs. You do this with the 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 the READ, DATA, and RESTORE statements, or you can name constants with the known values.
- You can read data from files outside the program. You do this with the INPUT #, INPUT LINE #, and LINPUT # statements.

The following sections describe how to use these statements in detail.

5.1.1 Providing Input Interactively

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

5.1.1.1 INPUT Statement

The INPUT statement interactively prompts the user for data. You can use the optional prompt string to clarify the input request by specifying the type and number of data elements required by the program. This is especially useful when the program contains many variables, or when someone else is running your program. For example:

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

Output

```
PLEASE TYPE 3 INTEGERS? 25,50,75 Return
THEIR SUM IS 150
```

When your program runs, BASIC stops at each INPUT, LINPUT, or INPUT LINE statement, prints a string prompt, if specified, and an optional question mark (?)[1] followed by a space; it then waits for your input. By using either a comma or semicolon, you can affect the format of your string prompt as follows:

- If you have a semicolon separating the input prompt string from the variable, BASIC prints the question mark and space immediately after the input prompt string.
- If you have a comma separating the input prompt string from the variable, BASIC prints the input prompt string, skips to the next print zone, and then prints the question mark and space.

See Section 5.2.1 for more information about print zones. For more information about formatting string prompts, see Section 5.1.1.3.

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

```
INPUT A,B
END
```

Output

```
? 5 Return
? 6 Return
```

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 Return
?   Return
```

These responses assign the value 5 to variable *A* and zero to variable *B*. In contrast, if you provide more values than there are variables, BASIC ignores the excess.

In the following example, BASIC ignores the extra value (8). You can type multiple values if you separate them with commas. Because commas separate variables in the PRINT statement, BASIC prints each variable at the start of a print zone.

[1] The SET NO PROMPT statement turns off the optional question mark; see Section 5.1.1.3.

```
INPUT A,B,C
PRINT A,B,C
END
```

Output

```
? 5,6,7,8 Return

 5            6            7
```

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. Because digits and a decimal point are valid text characters, numbers can be interpreted as strings. For example:

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

Output

```
Please type a number? 25.5
25.5
```

BASIC interprets the response as a 4-character string instead of as a numeric value.

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 or use the INPUT LINE or LINPUT statement. 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 leave out the end quotation mark, BASIC signals "Data format error" (ERR=50).

5.1.1.2 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, which are characters that the INPUT statement interprets as delimiters.

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.

In the following example, because both INPUT LINE and LINPUT treat your input as a string literal, BASIC interprets quotation marks, commas, and semicolons as characters, not as string delimiters. When *A$* is input with the INPUT LINE statement, the carriage return line terminator is stored as part of the string. The first PRINT statement tells BASIC to print all three variables on one line, starting each one in a new print zone. However, when BASIC prints the three strings, it prints the carriage return character at the end of string *A$*; this terminates the current line and causes *B$* to begin on a new line.

```
INPUT LINE A$
LINPUT B$
LINPUT C$
PRINT A$, B$, C$
PRINT "DONE"
END
```

Output

```
? SINGLE, DOUBLE Return
? "GFLOAT" Return
? HFLOAT; REAL Data Types Return

SINGLE, DOUBLE
"GFLOAT"     HFLOAT; REAL Data Types
DONE
```

The INPUT, INPUT LINE, and LINPUT statements can accept data from a terminal or a terminal-format file. See Section 5.3 for information about I/O to terminal-format files.

5.1.1.3 Enabling and Disabling the Question Mark Prompt

With the SET PROMPT statement, HP BASIC allows you to enable and disable the question mark prompt.

By default, HP BASIC displays the question mark prompt. The following example displays the default prompt string:

```
INPUT "Please input 3 integer values";A%, B%, C%
```

Output

```
Please input 3 integer values?
```

You can, however, disable the question mark prompt by specifying the SET NO PROMPT statement.

```
SET NO PROMPT
INPUT "Please input 3 integer values";A%, B%, C%
```

Output

```
Please input 3 integer values
```

When you disable the question mark prompt, you can specify your own prompt at the end of each prompt string. The following example inserts a colon at the end of the prompt string:

```
SET NO PROMPT
INPUT "Please enter your name: ";Employee_name$
```

Output

```
Please enter your name:
```

Now, if the SET PROMPT statement is specified, BASIC displays both the colon and a question mark.

```
SET PROMPT
INPUT "Please enter your name: ";Employee_name$
```

Output

```
Please enter your name: ?
```

The SET [NO] PROMPT statement is valid for INPUT, LINPUT, INPUT LINE, and MAT INPUT statements. If the prompt is disabled, any one of the following commands reenables it:

- The SET PROMPT statement
- The CHAIN statement
- The NEW, OLD, RUN, or SCRATCH compiler command

5.1.2 Providing Input from the Source Program

The following sections describe the READ, DATA, and RESTORE statements. 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.

5.1.2.1 READ and DATA Statements

The READ statement reads values from a data block. A data pointer keeps track of the data read. Each time the READ statement requests data, BASIC retrieves the next available constant from a DATA statement. The DATA statement contains the values that the READ statement reads. In a DATA statement, integer constants are whole numbers; they cannot be followed by a percent sign. In the following example, BASIC signals an error because the integer constants in the DATA statement contain percent signs:

```
10    WHEN ERROR USE catch_it
         DATA 1%, 2%, 3%
20       READ A%, B%, C%
      END WHEN
400   HANDLER catch_it
         PRINT "ERROR NUMBER IS "; ERR
         PRINT "ERROR AT LINE "; ERL
         PRINT "ERROR MESSAGE IS "; ERT$(ERR)
      END HANDLER
500   END
```

Output

```
ERROR NUMBER IS 50
ERROR AT LINE 20
ERROR MESSAGE IS %Data format error
```

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".

READ statements can appear either before or after their corresponding DATA statements. The only restriction is that the DATA statements must be in the same order as their corresponding READ statements.

You can have more than one DATA statement in a program. DATA statements are ignored without at least one READ statement. You can use an ampersand to continue a DATA statement. For example:

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

Comment fields are not allowed in DATA statements. For example, the following statements cause *A$* to contain the string "ABC !COMMENT":

```
READ A$
DATA 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 do one of the following:

- Assign alphabetic characters to a numeric variable. BASIC signals "Data format error" (ERR=50).
- Have more variables in the READ statements than there are values in the DATA statements. BASIC signals "Out of data" (ERR=57).

BASIC ignores excess data in DATA statements.

The following example of READ and DATA mixes string and floating-point data types. The first READ statement reads the first data item in the program: "The circumference is". The second READ statement reads the second data item: 40.5.

```
DATA "The circumference is"
DATA 40.5
READ text$
READ radius
CIRCUMFERENCE = PI * radius * 2
PRINT text$; CIRCUMFERENCE
END
```

Output

```
The circumference is 254.469
```

5.1.2.2 RESTORE Statement

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

RESTORE resets the data pointer to the beginning of the first DATA statement in the program unit. You can then read data values again. Consider the following 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 statement (line 30), the first three values are read again:

E=6
F=3
G=4

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

E=7
F=9
G=2

5.2 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. For information about the PRINT USING statement, see Chapter 14.

When you use the PRINT statement, HP BASIC does the following:

- Precedes positive numbers with a space and negative numbers with a minus sign
- Prints a space after every number
- Prints strings without 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:

```
A = 45
B = 55
PRINT A + B
END
```

Output

```
 100
```

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

```
A = 45
B = 55
PRINT "A + B"
END
```

Output

```
A + B
```

The PRINT statement without an expression prints a blank line.

```
PRINT "This example leaves a blank line"
PRINT
PRINT "between two lines."
END
```

Output

```
This example leaves a blank line

between two lines.
```

5.2.1 Print Zones—The Comma and the 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. For example:

```
INPUT A ,B ,C ,D ,E ,F
PRINT A ,B ,C ,D ,E ,F
END
```

Output

```
? 5,10,15,20,25,30 Return
 5             10            15            20            25
 30
```

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

```
A = 5
B = 10
PRINT "first zone",,"third zone"
PRINT A,,B
END
```

Output

```
first zone                  third zone
 5                           10
```

If you separate print elements with a semicolon, BASIC does not move to the next print zone. In the following example, the first PRINT statement prints two numbers. (Printed numbers are preceded by a space or a minus sign and followed by one space.) The second PRINT statement prints two strings.

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

Output

```
 10  20
ABCXYZ
```

Whether you use a comma or a semicolon at the end of the PRINT statement, the cursor remains at its current position until BASIC encounters another PRINT or INPUT statement. In the following example, BASIC prints the current values of *X*, *Y*, and *Z* on one line because a comma follows the last item in the line PRINT X, Y:

```
INPUT X,Y,Z
PRINT X,Y,
PRINT Z
END
```

Output

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

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

```
!No comma after I%, so each element
!Prints on its own line
!
PRINT I% FOR I% = 1% TO 10%
PRINT
!
!A comma follows J%, so each
!element prints in a separate zone
!
MARGIN 80%
PRINT J%, FOR J% = 1% TO 10%
PRINT
```

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

Output

```
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. For example:

```
PRINT "first zone",,"third zone",,"fifth zone"
END
```

Output

```
first zone                third zone                fifth zone
```

The extra comma between strings causes BASIC to skip another print zone. In the following example, the first string is longer than the print zone. When the two strings are printed, the second string begins in the third print zone because that is the next available print zone after the first string is printed.

```
PRINT "abcdefghijklmnopqrstuvwxyz","pizza"
PRINT "first zone","second zone","third zone"
```

Output

```
abcdefghijklmnopqrstuvwxyz  pizza
first zone    second zone   third zone
```

5.2.2 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:

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

Output

```
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.
- 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.
- When you print DECIMAL values, HP BASIC prints up to 31 digits.

HP BASIC follows these rules for printing floating-point numbers:

- If a floating-point number can be represented exactly by 6 decimal digits (or fewer) and, optionally, a decimal point, BASIC prints it that way.
- When you print a floating-point number whose integer portion is 6 decimal digits or less (for example, 1234.567), BASIC rounds the number to 6 digits (1234.57). If the integer portion of the number is 7 decimal digits or larger, BASIC rounds the number to 6 digits and prints it in E format. See the *HP BASIC for OpenVMS Reference Manual* for more information about E format.
- When you print a floating-point number with magnitude from 0.1 to 1, BASIC rounds it to 6 digits. When you print a floating-point number with more than 6 digits, and with magnitude smaller than 0.1, BASIC rounds it to 6 digits and prints it in E format.

The PRINT statement displays only up to 6 digits of precision for floating-point numbers. This corresponds to the precision of the SINGLE or SFLOAT data types. To display the extra digits in DOUBLE, GFLOAT, TFLOAT, or XFLOAT numbers, you must use the PRINT USING statement. See Chapter 14 for more information about the PRINT USING statement.

The following example shows how BASIC prints various numbers with single precision:

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

Output

```
 .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
```

5.3 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 useful for creating files to be printed on a line printer, 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. For more information, see Chapter 13.

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.

5.3.1 Opening and Closing a Terminal-Format File

You use the OPEN statement to create a file, or to gain access to an existing file. If you do not specify either FOR INPUT or FOR OUTPUT in the OPEN statement, 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.

5.3.2 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:

```
PRINT "This program creates a daily sales report file named SALES.DAT"
OPEN "SALES.DAT" FOR OUTPUT AS FILE #4%
PRINT #4%, "Salesperson","Sales Area","Items Sold"
PRINT #4%
INPUT "How many salespersons for today's report"; sales_persons%
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%
CLOSE #4%
END
```

Output

```
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
```

This program first prints a header explaining its purpose, then opens a terminal-format file on channel 4. After this file is opened, the two PRINT # statements place an explanatory header followed by a blank line into the file.

The program then prompts you for the number of salespersons for which data is to be entered. The FOR...NEXT loop prompts for the name, sales area, and items sold for each salesperson. The FOR...NEXT loop executes only as many times as there are salespersons. See Chapter 9 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 3, 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 DCL command TYPE, you see that the information is printed under the proper headers. You can also print the file on a line printer. The PRINT # statement formats the output in print zones as the PRINT statement does.

```
$ TYPE SALES.DAT

Salesperson   Sales Area    Items Sold

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

6
Arrays

An array is a set of data that is ordered in any number of dimensions. This chapter describes how to create and use HP BASIC arrays.

6.1 Overview of Arrays

A one-dimensional array is called a list or vector. A two-dimensional array is called a matrix. HP BASIC arrays can have up to 32 dimensions, and a specific type of HP BASIC arrays can be redimensioned at run time. In addition, you can specify the data type of the values in an array by using data type keywords or suffixes.

The subscript of an element in an array defines that element's position in the array. When you create an array, you specify:

- The number of dimensions that the array contains
- The range of values for the subscripts in each dimension of the array

BASIC arrays are zero-based by default; that is, when calculating the number of elements in a dimension, you count from zero to the number of elements specified. For example, an array with an upper bound of 10 and no specified lower bound has 11 elements: 0 to 10, inclusive. The array *My_array*(3,3) has 16 elements: 0 to 3 in each dimension, or 4^2.

BASIC also lets you specify a lower bound for any or all dimensions in an array, unless the array is a virtual array. By specifying lower and upper bounds for arrays, you can make your array subscripts meaningful. For example, the following array contains sales information for the years 1990 to 1999:

```
DECLARE REAL Sales_data(1990 TO 1999)
```

To refer to an element in the array *Sales_data*, you need only specify the year you are interested in. For example, to print the information for the year 1999, you would enter:

```
PRINT Sales_data(1999)
```

You can create arrays either implicitly or explicitly. You implicitly create arrays having any number of dimensions by referencing an element of the array. If you implicitly create an array, BASIC sets the upper bound to 10 and the lower bound to zero. Therefore, any array that you create implicitly contains 11 elements in each dimension.

The following example refers to the array *Student_grades*. If the array has not been previously declared, BASIC will create a one-dimensional array with that name. The array contains 11 elements.

```
Student_grades(8) = "B"
```

You create arrays explicitly by declaring them in a DIM, DECLARE, COMMON, or MAP statement, or record declaration. Note that if you want to specify lower bounds for your array subscripts, you must declare the array explicitly.

When you declare an array explicitly, the value that you give for the upper bound determines the maximum subscript value in that dimension. If you specify a lower bound, then that is the minimum subscript value in that dimension. If you do not specify a lower bound, BASIC sets the lower bound in that dimension to zero. You can specify bounds as either positive or negative values. However, the lower bound of each dimension must always be less than or equal to the upper bound for that dimension.

You can use MAT statements to create and manipulate arrays; however, MAT statements are valid only on arrays of one or two dimensions. In addition, the lower bounds of all dimensions in an array referenced in a MAT statement must be zero.

6.2 Creating Arrays Explicitly

You can create arrays explicitly with four BASIC statements: DECLARE, DIMENSION, COMMON, and MAP.

In addition, you can declare arrays as components of a record data type. See Chapter 8 for more information about records.

Normally, you use the DECLARE statement to create arrays. However, you might want to create the array with another BASIC statement as follows:

- Use the DIM statement to create virtual arrays and arrays that can be redimensioned at run time.
- Use the COMMON statement to create arrays that can be shared among program modules or to create arrays of fixed-length strings.

- 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, the bounds you specify determine the array's size. The maximum value allowed for a bound can be as large as 2147483467; however, this number is actually limited by the amount of virtual storage available to you. Very large arrays and arrays with many dimensions can cause fatal errors at both compile time and run time.

The following restrictions apply to arrays:

- When referencing an array, you must use the same number of subscripts as was specified when the array was created.
- You can use identical names for a simple variable and an array; for example, A% and A%(5,5). However, this is not a 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 error "Inconsistent subscript usage" at compile time.
- If subscript checking is enabled, HP BASIC signals the error "Subscript out of range" (ERR=55) if you reference an array element whose subscripts are one of the following:
  - Greater than the current upper bound of the array
  - Less than the current lower bound of the array
  - Less than zero where no lower bound was specified

6.2.1 Creating Arrays with the DECLARE Statement

The DECLARE statement creates and names variables and arrays. All elements of arrays created with the DECLARE statement are initialized to zero or the null string. The following statement creates a longword integer array with 11 elements:

```
DECLARE LONG FIRST_ARRAY(1980 TO 1990)
```

Note that the STRING data type with the DECLARE statement 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 or as part of a RECORD structure.

6.2.2 Creating Arrays with the DIM Statement

The DIM statement creates and names one or more arrays. Use the DIM statement to create an array when you want to:

- Redimension the array at run time
- Create a virtual array

When creating arrays with the DIM statement, you specify the data type of the array elements with a data type keyword, a special suffix on the array name, or both. 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 with a dollar sign ($), the array stores string data.
- If the array name ends with a percent sign (%), the array stores integer data.
- If the array name does not end with either a percent sign or 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 4 for more information about default data types.

Even if the DIM statement contains a data type keyword, the array name can still end in the appropriate data type suffix. This makes the data type of the array immediately obvious.

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 statements reinitialize the array does not matter. See Section 6.6 for more information about MAT statements.

6.2.2.1 Declarative DIM Statements

Declarative DIM statements have integer constants as bounds. The percent sign is optional for bounds; however, BASIC signals the error "Integer constant required" if a constant bound contains a decimal point. The following statement creates a 101-element virtual array containing string data. The elements of this array can each have a maximum length of 256 characters.

```
DIM #1%, STRING VIRT_ARRAY(100) = 256%
```

The following restrictions apply to the use of declarative DIM statements:

- A declarative DIM statement must lexically precede any reference to the array it dimensions.
- The lower bounds of all virtual array dimensions must be zero.
- You must open a VIRTUAL file on the specified channel before you can access elements of the virtual array.

6.2.2.2 Executable DIM Statements

Executable DIM statements have 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 bounds of the array can become larger or smaller, but the number of dimensions cannot change. For example, 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.

Whenever an executable DIM statement executes, it reinitializes the array. If you change the values of an executable DIM statement, the initial values are reset each time the DIM statement is executed.

In the following example, the second DIM statement reinitializes the array *real_array*; therefore, *real_array(1%)* equals zero in the second PRINT statement:

```
X% = 10%
Y% = 20%
DIM real_array(X%)
real_array(1%) = 100
PRINT real_array(1%)
DIM real_array(Y%)
PRINT real_array(1%)
END
```

Output

```
100
0
```

You cannot reference an array named in an executable DIM statement until after the DIM statement executes. If you reference an array element declared in an executable DIM statement whose subscripts are larger than the bounds specified in the last execution of the DIM statement, BASIC signals the run-time error "Subscript out of range" (ERR = 55), provided subscript checking is enabled.

6.2.3 Creating Arrays with the COMMON Statement

Create arrays with the COMMON statement when you need an array of fixed-length strings, or when you want to share an array among program modules. Program modules can share arrays in COMMON statements by defining a common block with the same name.

The COMMON statements in the following programs create a 100-element array of fixed-length strings, each element 10 characters long. Because the main program and subprograms use the same common name, the storage for these arrays is overlaid when the programs are linked; therefore, both programs can read and write data to the array.

```
!Main Program
COMMON (ABC) STRING access_list(1 TO 100) = 10

!Subprogram
SUB SUB1
COMMON (ABC) STRING new_list(1 TO 100) = 10
```

6.2.4 Creating Arrays with the MAP Statement

Create arrays with the MAP statement 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.

You associate the array with a record buffer by naming the map in the MAP clause of the OPEN statement.

In the following example, 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.

```
MAP (ABC) STRING team(10) = 20, WORD bowling_scores(0 TO 32)
OPEN "BOWLING.DAT" AS FILE #1%, SEQUENTIAL VARIABLE, MAP ABC
```

6.3 Creating Arrays Implicitly

Create arrays implicitly as follows:

- By referencing an element of an array that has not been explicitly declared
- By using MAT statements

When you first create an implicit array, the lower bound is zero and the upper bound is 10. An array created by referencing an element can have up to 32 dimensions in 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 better programming practice to declare all arrays explicitly 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 (0 TO 10), (0 TO 10, 0 TO 10), (0 TO 10, 0 TO 10, 0 TO 10), and so on, depending on the number of dimensions specified in the array reference. For example, the following program implicitly creates three arrays and assigns a value to one element of each:

```
LET A(5,5,5) = 3.14159
LET B%(3) = 33
LET C$(2,2) = "Russell Scott"
END
```

The first LET statement creates an 11-by-11-by-11 array that stores floating-point numbers and assigns the value 3.14159 to element (5,5,5). The second LET statement creates an 11-element list that stores integers and assigns the value 33 to element (3), and the third LET statement creates an 11-by-11 string array and assigns the value "Russell Scott" to element (2,2).

When you create an implicit numeric array by referring to 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 implicitly create an array, you cannot specify a subscript greater than 10. An attempt to do so causes BASIC to signal "Subscript out of range" (ERR = 55), provided that subscript checking is enabled.

Note that you cannot create an array implicitly, then redimension the array with an executable DIM statement. The DIM statement must execute before any reference to the array.

An array name cannot appear in a declarative statement after the array has been implicitly declared by a reference. The following DECLARE statement is therefore illegal and causes HP BASIC to signal the compile-time error "illegal multiple definition of name NEW_ARRAY."

```
new_array (5,5,5) = 1
DECLARE LONG new_array (15,10,5)
```

6.4 Determining the Bounds of an Array

BASIC provides two built-in functions, LBOUND and UBOUND, that allow you to determine the lower and upper bounds, respectively, for any dimension in an array.

The following example sets up four variables that contain the lower and upper bounds of both dimensions of the array *Sales_data*. These variables represent the years and months for which there is sales data available. The two FOR...NEXT loops print all the sales information in the array, starting with the first year and month, and ending with the last year and month.

```
DECLARE Sales_data(1900 TO 1999, 1 TO 12)

Month_start% = LBOUND (Sales_data, 2)
Year_start% = LBOUND (Sales_data, 1)
Month_end% = UBOUND (Sales_data, 2)
Year_end% = UBOUND (Sales_data, 1)
FOR Year% = Year_start% TO Year_end%

    FOR Month% = Month_start% TO Month_end%
        PRINT Sales_data(Year%, Month%)
    NEXT Month%

NEXT Year%
```

Note

You cannot implicitly declare arrays with the LBOUND and UBOUND functions. These functions can be used only with arrays that have been previously declared.

6.5 Assigning and Displaying Array Values

The following sections explain how to access and write to BASIC arrays with the LET and PRINT statements.

6.5.1 Assigning Values with the LET Statement

The LET statement assigns values to individual array elements. For example:

```
DIM voucher_num%(100)
   .
   .
   .
LET voucher_num%(20) = 3253%
   .
   .
   .
END
```

You can also assign values to a portion of an array with the LET statement and a FOR...NEXT loop. In the following example, the FOR...NEXT loop assigns zero to array elements (1,5) to (1,10), (2,5) to (2,10), and (3,5) to (3,10):

```
DIM po_number%(100,100)
   .
   .
   .
FOR I% = 1% TO 3%
    FOR J% = 5% TO 10%
          LET po_number%(I%,J%) = 0%
    NEXT J%
NEXT I%
   .
   .
   .
END
```

6.5.2 Listing Array Elements with the PRINT Statement

You print individual array elements by naming those elements in the PRINT statement. For example:

```
PRINT parts_list$(35%)
```

With a FOR...NEXT loop, you can print all or part of an array. For example:

```
DIM capture_ratio(10,10)
   .
   .
   .
FOR Y% = 7% TO 10%
  FOR X% = 7% TO 10%
      PRINT capture_ratio(X%,Y%)
  NEXT X%
NEXT Y%
```

6.6 Using MAT Statements

Note

The MAT statements discussed in this section are not related to the MAT GRAPH and MAT PLOT graphics statements. For more information about these statements, see *Programming with VAX BASIC Graphics*.

MAT statements let you assign values to or display entire arrays with a single statement. They also let you do the following:

- 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

MAT statements are valid only on arrays of one or two dimensions. 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 of your arrays, and you should not depend on data in these elements if your program uses MAT statements.

Note

MAT statements cannot be used with arrays that have lower bounds other than zero. An attempt to specify a lower bound other than zero for an array in a MAT statement results in a compile-time error.

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 6–1 lists MAT statements and explains their functions.

Table 6–1 MAT Statements

Statement	Function
MAT	Assigns values of zero, 1, or a null string to array elements. Also copies the values of one array to another and performs matrix arithmetic.
MAT READ	Assigns DATA statement values to array elements.
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.

In the following example, the first MAT statement creates the string array *z_array$* with eight rows and eight columns and assigns a null string to all

elements. The second MAT statement redimensions the array to six rows and six columns. The third MAT statement adds the values in each corresponding element of arrays *B* and *C* and stores the values in the corresponding elements of array *A*.

```
MAT z_array$ = NUL$(7,7)
MAT z_array$ = NUL$(5,5)
MAT A = B + C
END
```

6.6.1 MAT Statement

The MAT statement can create an array and optionally assign values to all elements in that array. By specifying one of the MAT statement keywords, you can initialize arrays in one of four ways. Table 6–2 lists the MAT statement keywords and their functions.

Table 6–2 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 1, except those in row and column zero.
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 1. IDN applies to square arrays only.
NUL$	Sets the value of all elements in a string array to the null string, except those in row and column zero.

The array name can specify an existing array. MAT statements do not assign values to row and column zero.

Note that the MAT statement does not require subscripts. In the case of 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 you are 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. Consider the following example:

```
DIM A(10,10), B(15), C(20,20)
MAT A = ZER             !Sets all elements of A to 0
MAT B = CON(10)         !Sets elements of B to 1; redimensions B
MAT C = IDN(10,10)      !Redimensions C to 10x10 identity matrix
PRINT "ARRAY A:"
MAT PRINT A;
PRINT
PRINT "ARRAY B:"
MAT PRINT B;
PRINT
PRINT "ARRAY C:"
MAT PRINT C;
```

Output

```
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
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 B:
1  1  1  1  1  1  1  1  1  1

ARRAY C:
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
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
```

6.6.2 MAT READ Statement

The MAT READ statement assigns values from DATA statements to array elements. Subscripts define either the dimensions of the array being created or the new dimensions of an existing array; subscripts are optional in MAT READ statements.

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.

In the following example, 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:

```
MAT READ B(2,2)
MAT READ C(2,2)
PRINT
PRINT "MATRIX B"
PRINT
PRINT
MAT PRINT B;
PRINT
PRINT "MATRIX C"
PRINT
PRINT
MAT PRINT C;
DATA 1,2,3,4,5,6,7,8,9,10
END
```

Output

```
MATRIX B

 1  2
 3  4
MATRIX C

 5  6
 7  8
```

6.6.3 MAT INPUT [#] Statement

The MAT INPUT statement assigns values from your terminal to array elements. The MAT INPUT statement reads data from a terminal-format file and writes it to an array. The optional subscripts in a MAT INPUT statement define either the dimensions of the array being created implicitly or the new dimensions of an existing array. If you are implicitly 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 that you can disable with the SET NO PROMPT statement and then enable with the SET PROMPT statement. However, you cannot include a string prompt with the MAT INPUT statement.

When you enter a series of values separated by commas, BASIC enters the values you supply 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. If you provide more items than there are elements, BASIC ignores the excess.

The MAT INPUT statement takes values from an open file and assigns them to the matrix elements by rows, starting with element (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, multiple values must be separated with commas.

In the following example, the open file on channel 3 contains the following data: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13. The MAT INPUT statement reads this data and uses it to fill the array *A*, filling in row 1 before beginning row 2. The MAT INPUT B(2,2) statement dimensions array *B* to 9 elements (0 to 2 in each dimension) and provides values for all the elements except those in row and column zero.

```
MAT INPUT #3, A
PRINT
MAT PRINT A;
MAT INPUT B(2,2)
PRINT
MAT PRINT B;
```

Output

```
  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
```

Note that the MAT PRINT statement does not print row and column zero. For more information about the MAT PRINT statement, see Section 6.6.5.

The MAT INPUT statement can also redimension an existing array.

```
DIM new_array%(5,5)
MAT INPUT new_array%(2,4)
MAT PRINT new_array%;
END
```

Output

```
 ? 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.

6.6.4 MAT LINPUT [#] Statement

The MAT LINPUT statement assigns string values to string array elements. The MAT LINPUT statement reads string values from a terminal-format file and writes them to a string array.

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 supply (including commas, semicolons, and quotation marks, but excluding the line terminator) to an array element.

```
DIM emp_nam$(5,5)
MAT LINPUT emp_nam$(2,2)
PRINT emp_nam$(1,1)
PRINT emp_nam$(1,2)
PRINT emp_nam$(2,1)
PRINT emp_nam$(2,2)
END
```

Output

```
? SMITH
? JONES
? WHITE
? BLACK
SMITH
JONES
WHITE
BLACK
```

By specifying the subscripts (2,2), MAT LINPUT redimensions the array to nine elements and overwrites the old values (assigning the values in the same manner as MAT INPUT; see Section 6.6.3). BASIC then prompts for these elements.

MAT LINPUT also excludes line terminators when assigning values to string array elements. MAT LINPUT places the values from the open file into the specified array, filling the array by rows, starting with element (1,1). If there are more values in the file than there are array elements, BASIC ignores the excess records. If there are fewer, BASIC assigns a null string to the remaining elements.

The following program reads 50 records from the open disk file and assigns them to the array named *part_name$*. If there are more than 50 records in the file, BASIC ignores the excess records. If there are fewer than 50 records, then BASIC fills the remaining elements of the array with the null string.

```
DIM part_name$(50)
MAT LINPUT #1%, part_name$
```

6.6.5 MAT PRINT [#] Statement

The MAT PRINT statement prints some or all of an array's elements, excluding row and column zero. The MAT PRINT # statement takes values from an array by row, starting with element (1,1), and writes each element to a sequential record in the terminal-format file.

Subscripts are optional in MAT PRINT statements. If you do not specify subscripts, MAT PRINT displays the entire array, excluding row and column zero. If you specify subscripts, MAT PRINT displays the specified subset of the array. In the case of the MAT PRINT # statement, the subscripts determine how many array elements are written to the file. 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 row on a separate line. Data values on each line are packed together with no intermediate spaces. However, if the last character in the MAT PRINT [#] array list is a 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 a semicolon after the array name, BASIC prints each array element on a separate line. In the following example, the first MAT PRINT statement does not end in a comma or semicolon, so each element is printed on a separate line. The second MAT PRINT statement prints the elements twice, the first time starting each element in a new print zone, and the second time leaving a space before and after each value. The MAT PRINT # statement sends the last two lines of output to a terminal-format file.

```
MAT INPUT A(5)
PRINT
MAT PRINT A
PRINT
MAT PRINT A, A;
MAT PRINT #3, A, A;
END
```

Output

```
? 5

  5
  0
  0
  0
  0

  5          0           0           0           0

  5  0  0  0  0
```

6.6.6 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 or when there are fewer data items than array elements to contain them.

BASIC provides two functions that let you determine how much data the MAT statements transfer: NUM and NUM2.

For two-dimensional arrays, the NUM function returns an integer value specifying the row number of the last data item transferred, and the NUM2 function returns an integer value specifying the column number of the last data item transferred. For one-dimensional arrays, the NUM function returns the number of items entered, and the NUM2 function returns a zero.

With these functions, you can determine the number of items transferred from a terminal-format file. Note, however, that you cannot use the NUM and NUM2 functions to implicitly declare an array. In the following example, the terminal-format file EMP.DAT contains the values 1 to 17, inclusive. When these values are read with the MAT INPUT # statement, NUM and NUM2 represent the row and column number, respectively, of the last value read.

```
OPEN "EMP.DAT" FOR INPUT AS FILE #3%
DIM emp_name$(5,5)
MAT INPUT #3%, emp_name$
PRINT NUM, NUM2
END
```

Output

```
 4               2
```

6.7 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. Note that 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.

6.7.1 Arithmetic Matrix Operations

MAT operators perform matrix assignment, addition, subtraction, and 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.

6.7.1.1 Assignment

You can assign all values in one array to another array with the MAT statement. In the following example, each element of *new_array* is set to the corresponding element in *old_array*. The dimensions of *new_array* are also redimensioned to the dimensions of *old_array*.

```
MAT new_array = old_array
```

6.7.1.2 Addition and Subtraction

You can add the elements of two arrays. In the following statement, 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.

```
MAT sum_list% = first_list% + second_list%
```

You can also subtract the elements of two arrays. The following program subtracts one array from another:

```
DIM first_array(30,30)
DIM second_array(30,30)
DIM difference_array(30,30)
   .
   .
   .
MAT difference_array = first_array - second_array
```

Each element of *difference_array* is the arithmetic difference of the corresponding elements of the input arrays.

6.7.1.3 Multiplication

You can multiply the elements of two arrays, provided that the number of columns in the first array equals the number of rows in the second array. The resulting array contains the dot product of the two input arrays.

```
DIM A(2,2), B(2,2), C(2,2)
A(1,1) = 1
A(1,2) = 2
A(2,1) = 3
A(2,2) = 4
B(1,1) = 5
B(1,2) = 6
B(2,1) = 7
B(2,2) = 8
MAT C = A * B
MAT PRINT C

 19
 22
 43
 50
```

You can also multiply a matrix by a scalar quantity. 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. The following example multiplies the elements of *inch_array* by the inch-to-centimeter conversion factor and places these values in *cm_array*:

```
DIM inch_array(5), cm_array(5)
MAT READ inch_array
DATA 1,12,36,100,39.37
MAT cm_array = (2.54) * inch_array
MAT PRINT cm_array,
END
```

Output

```
 2.54     30.48     91.44     254     99.9998
```

6.7.2 Matrix Functions

BASIC provides the following matrix functions:

TRN
INV
DET

With these functions, you can transpose and invert matrices and find the determinant of an inverted matrix.

6.7.2.1 TRN Function

The TRN function transposes a matrix. 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.

The following example creates a 3-by-5 matrix, transposes it, and prints the results:

```
DIM B(3,5)
MAT READ B
MAT A = TRN(B)
DATA 1,2,3,4,5
DATA 6,7,8,9,10
DATA 11,12,13,14,15
MAT PRINT B;
MAT PRINT A;
END
```

Output

```
 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
```

6.7.2.2 INV Function

The INV function inverts a matrix. BASIC can invert a matrix only if its subscripts are identical and it can be reduced to the identity matrix by elementary row operations. The input matrix multiplied by the output matrix (its inverse) always gives the identity matrix as a result.

```
MAT INPUT first_array(3,3)
MAT PRINT first_array;
PRINT
MAT inv_array = INV (first_array)
MAT PRINT inv_array;
PRINT
MAT mult_array = first_array * inv_array
MAT PRINT mult_array;
PRINT
D = DET
PRINT D
```

Output

```
? 4,0,0,0,0,2,0,8,0
 4  0  0
 0  0  2
 0  8  0

 .25  0  0
 0  0  .125
 0  .5  0

 1  0  0
 0  1  0
 0  0  1

 -64
```

6.7.2.3 DET Function

The DET function returns the determinant of a matrix. 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.

7
Data Definition

This chapter briefly describes how to define program objects, explicitly assign data types, and allocate and use data storage.

7.1 Declarative Statements

You use **declarative statements** to define objects in a HP 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. Declarative statements can also be used to define user-defined data types (RECORD statements). See Chapter 8 for more information about the RECORD statement.

You use declarative statements to assign data types to:

- Variables
- Arrays
- Named constants
- Values returned by functions

By declaring the objects used in your program, you make the program easier to understand, modify, and debug.

7.2 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 how the computer should interpret these patterns.

HP BASIC programs allow five general data types: integer, floating-point, string, packed decimal, and record. Each data type is suited for a particular type of task. For example, integers are useful for numeric computations involving whole numbers, strings provide a way to manipulate alphanumeric

characters, and packed decimal data is useful for manipulating numeric values that require precise representation.

For more information about HP BASIC data types, see the *HP BASIC for OpenVMS Reference Manual*.

7.3 Setting the Default Data Type and Size

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

- With the OPTION statement
- With the following qualifiers:
  - /TYPE_DEFAULT
  - /INTEGER_SIZE
  - /REAL_SIZE
  - /DECIMAL_SIZE

The OPTION statement can override the defaults set with qualifiers. For example, the following statement sets the default integer type to be LONG:

```
OPTION SIZE = INTEGER LONG
```

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 the OPTION statement can also specify the following:

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

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

The OPTION statement in the following example specifies that all program variables must be explicitly typed and that all implicitly typed constants are INTEGER. In addition, any variable typed as INTEGER is a LONG integer and any variable typed as REAL is a DOUBLE floating-point number.

```
OPTION  TYPE = EXPLICIT,         ! Variables must be declared         &
        CONSTANT TYPE = INTEGER, ! All implicit constants be integers &
        SIZE = INTEGER LONG,     ! 32-bit integers by default         &
        SIZE = REAL DOUBLE       ! 64-bit floating-point
                                 ! numbers by default
```

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

7.4 Declaring Variables

A variable is a named quantity whose value can change during program execution. Variables may be implicitly or explicitly declared. HP BASIC accepts the following types of variables:

- Floating-point
- Integer
- String
- RFA
- Packed decimal
- Record

For more information about declaring variables, see the *HP BASIC for OpenVMS Reference Manual*.

7.5 Declaring Named Constants

A constant is a value that does 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, BASIC provides notation for binary, octal, decimal, and hexadecimal constants.

For more information about named constants, see the *HP BASIC for OpenVMS Reference Manual*.

7.6 Operations with 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 can also be converted depending on the data type of the variable to which it is assigned.

When assigning values to variables, HP 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, HP BASIC signals "Integer error or overflow," "Floating-point error or overflow," or "DECIMAL error or overflow."

In general, HP 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. HP 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, HP BASIC converts the result to the data type of the variable. For more information about multiple data types, see the *HP BASIC for OpenVMS Reference Manual*.

7.7 Allocating Dynamic and Static Storage

HP BASIC programs allocate both dynamic and static storage. **Dynamic storage** is allocated when the program executes, whereas the size of **static storage** does not change during program execution.

Variables and arrays declared by the following means use dynamic storage:

- DECLARE statements
- DIMENSION statements
- Implicitly declared variables

Normally, string variables and arrays declared in these ways 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 Chapter 8 for more information about the RECORD statement.

Variables and arrays appearing in MAP or COMMON statements use static storage. Hence all string variables appearing in MAP or COMMON statements are fixed-length strings. MAP and COMMON statements create a named storage area called a program section, or 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, HP BASIC supplies a default PSECT name of $BLANK.

The remainder of this section explains how to use COMMON and MAP statements for static storage.

7.7.1 COMMON Statement

The COMMON statement defines a named area of storage (called a PSECT). Any HP BASIC subprogram can access the values in a common area by specifying a common with the same name. An item in a COMMON statement can be any one of the following:

- Numeric variable
- Numeric array
- Fixed-length string variable
- Array of fixed-length strings
- RECORD instance
- FILL item
- RFA 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 following statement specifies 2 bytes for *emp.code*, 3 bytes for *wage.code*, and 22 bytes for *dep.code*:

```
COMMON (code) STRING emp.code=2, wage.code=3, dep.code=22
```

In a single program module, multiple common areas with the same name allocate storage end-to-end in a single PSECT. That is, HP BASIC concatenates all common areas with the same name in the same program module, in order of appearance. For example, the following statements allocate storage for five LONG integers in a single PSECT named *into*:

```
COMMON (into) LONG call_count, sub1_count, sub2_count
COMMON (into) LONG sub3_count, sub4_count
```

When you explicitly declare an array, HP BASIC allows you to specify both upper and lower bound values. The value you supply as the upper bound determines the maximum subscript value for a given dimension, and the value you supply for the lower bound determines the minimum subscript value for a given dimension.

For more information about specifying bounds with the COMMON statement, see Chapter 6 and the *HP BASIC for OpenVMS Reference Manual*.

7.7.2 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.

A map item can be one of the following:

- Numeric variable
- Numeric array
- Fixed-length string variable
- Array of fixed-length strings
- RECORD instance
- FILL item

When you explicitly declare an array, HP BASIC allows you to specify both upper and lower bound values. The value you supply as the upper bound determines the maximum subscript value for a given dimension, and the value you supply for the lower bound determines the minimum subscript value for a given dimension.

For more information about specifying bounds with the MAP statement, see Chapter 6 and the *HP BASIC for OpenVMS Reference Manual*.

7.7.2.1 Single Maps

You associate a map with a record buffer by referencing the map in the OPEN statement.

The MAP statement must appear before any reference to map variables. 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 or UPDATE statement. For more information, see Chapter 13.

The following program example uses map variables to access fields in payroll records:

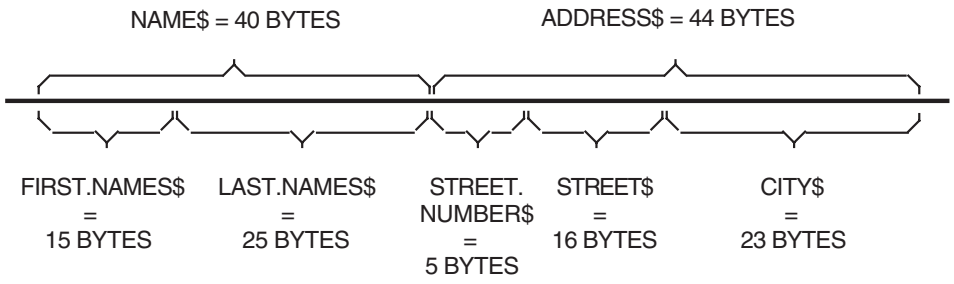
```
WHEN ERROR USE eof_handler
DECLARE INTEGER CONSTANT EOF = 11

MAP (PAYROL) STRING emp_name, LONG wage_class,     &
                      STRING sal_rev_date, SINGLE tax_ytd

OPEN "payroll.dat" FOR INPUT AS FILE #4%           &
                     ,ORGANIZATION SEQUENTIAL      &
                     ,ACCESS READ                  &
                     ,MAP PAYROL

OPEN "payrol.new" FOR OUTPUT AS FILE #5%           &
             ,ORGANIZATION SEQUENTIAL              &
             ,ACCESS WRITE                         &
             ,MAP payrol

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
END WHEN
HANDLER eof_handler
   IF ERR = EOF
     THEN
            PRINT "End of file"
     ELSE
            EXIT HANDLER
   END IF
END HANDLER
END
```

7.7.2.2 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. Figure 7–1 shows multiple maps and maps in use.

Figure 7–1 Multiple Maps



When you use more than one map to access a record buffer, HP 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 record size. For more information, see Chapter 13.)

You can also use multiple maps to interpret numeric data in more than one way. The following 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 to Z.

```
MAP (barray) BYTE alphabet(25)
MAP (barray) STRING ABC = 26
FOR I% = 0% TO 25%
    alphabet(I%) = I% + 65%
NEXT I%
PRINT ABC
END
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

7.7.3 FILL Items

FILL items reserve space in map and common blocks and in record buffers accessed by MOVE or REMAP statements. Thus, FILL items mask parts of the record buffer and let you skip over fields and reserve space in or between data elements.

FILL formats are available for all data types. Table 7–1 summarizes the FILL formats and their default allocations if no data type is specified.

Table 7–1 FILL Item Formats, Representations, and Default Allocations

FILL Format	Representation	Bytes Used
FILL	Floating-point	4, 8, 16, or 32
FILL(*n*)	*n* floating-point elements	4*n*, 8*n*, 16*n*, or 32*n*
FILL%	Integer (BYTE, WORD, LONG, or QUAD)	1, 2, 4, or 8
FILL%(*n*)	*n* integer elements	1*n*, 2*n*, 4*n*, or 8*n*
FILL$	String	16
FILL$(*n*)	*n* string elements	16*n*
FILL$ = *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 and optional data type-suffixes. The data-type and storage requirements are those of the last data type specified. For example:

```
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
- 24 bytes of padding
- One LONG variable, SSN
- 4 bytes of padding
- One REAL variable, SAL
- Space for five floating-point numbers (10, 20, or 80 bytes of padding, depending on the default size for floating-point numbers)

You can specify user-defined data types (RECORD names) for FILL items. In the following example, the first line defines a RECORD of data type *X*. The MAP statement contains a fill item of this data type, thus reserving space in the buffer for one RECORD of type *X*.

```
RECORD X
   REAL Y1, Y2(10)
END RECORD X
MAP (QED) X FILL
```

See Chapter 8 for more information about the RECORD statement.

7.7.4 Using COMMON and MAP Statements in Subprograms

The COMMON and MAP statements create a block of storage called a PSECT. This common or map storage block is accessible to any subprogram. A HP BASIC main program and subprogram can share such an area by referencing the same common or map name.

The following example contains common blocks that define:

- 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

```
!In a main program
COMMON (A1) STRING A, B = 10, LONG C
   .
   .
   .
!In a subprogram
COMMON (A1) STRING X, Z = 10, LONG Y
```

If a subprogram defines a common or map area with the same name as a common or map area in the main program, it overlays the common or map defined in the main program.

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. The following COMMON statements are in the same program module; therefore, they are concatenated in a single PSECT. The PSECT contains two 32-byte strings.

```
COMMON (XYZ) STRING A = 32
COMMON (XYZ) STRING B = 32
```

In contrast, the following COMMON statements are in different program modules, and thus overlay the same storage. Therefore, the PSECT contains one 32-byte string, called *A* in the main program and *B* in the subprogram.

```
!In the main program
COMMON (XYZ) STRING A = 32
   .
   .
   .
!In the subprogram
COMMON (XYZ) STRING B = 32
```

Although you can redefine the storage in a common section when you access it from a subprogram, you should generally 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, as shown in the following example:

```
COMMON (SHARE) WORD emp_num,              &
            DECIMAL (8,0) salary,         &
            STRING wage_class = 2
   .
   .
   .
!In the main program
%INCLUDE "COMMON.BAS"
   .
   .
   .
!In the subprogram
%INCLUDE "COMMON.BAS"
```

If you use the %INCLUDE directive, you can lessen the risk of a typographical error. For more information about using the %INCLUDE directive, see Chapter 16.

If you must redefine the variables in a PSECT, you should use the MAP statement or a record with variants for each overlay. When you use the MAP statement, use the %INCLUDE directive to create identical maps before redefining them, as shown in the following example. 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.

```
MAP (REDEF) STRING full_name = 40
   .
   .
   .
!In the main program
%INCLUDE "MAP.BAS"
   .
   .
   .
!In the subprogram
%INCLUDE "MAP.BAS"
MAP (REDEF) STRING first_name=15, MI=1, last_name=24
```

7.7.5 Dynamic Mapping

Dynamic mapping lets you redefine the position of variables in a static storage area. This storage area can be either a map name or a previously declared static string variable. Dynamic mapping requires the following HP BASIC statements:

- A declarative statement, such as a MAP statement, allocating a fixed-length storage area
- A MAP DYNAMIC statement, naming the variables whose positions can change at run time
- A REMAP statement, specifying the new positions of the variables named in the MAP DYNAMIC statement

The MAP DYNAMIC statement does not affect the amount of storage allocated. The MAP DYNAMIC statement causes HP BASIC to create internal pointers to the variables and array elements. Until your program executes the REMAP statement, the storage for each variable and each array element named in the MAP DYNAMIC statement starts at the beginning of the map storage area.

The MAP DYNAMIC statement is nonexecutable. With this statement, you cannot specify a string length. All string items have a length of zero until the program executes a REMAP statement.

The REMAP statement specifies the new positions of variables named in the MAP DYNAMIC statement. That is, it causes HP BASIC to change the internal pointers to the data. Because the REMAP statement is executable, it can redefine the pointer for a variable or array element each time the REMAP statement is executed.

With the MAP DYNAMIC statement, you can specify either a map name or a previously declared static string variable. When you specify a map name, a MAP statement with the same map name must lexically precede the MAP DYNAMIC statement.

In the following example, the MAP statement creates a storage area named *emp_buffer*. The MAP DYNAMIC statement specifies that the positions of variables *emp_name* and *emp_address* within the map area can be dynamically defined with the REMAP statement.

```
DECLARE LONG CONSTANT emp_fixed_info = 4 + 9 + 2
MAP (emp_buffer) LONG badge,                    &
                 STRING social_sec_num = 9,     &
                 BYTE name_length,              &
                      address_length,           &
                      FILL (60)

MAP DYNAMIC (emp_buffer) STRING emp_name,       &
                                emp_address

WHILE 1%
GET #1
REMAP (emp_buffer) STRING FILL = emp_fixed_info,            &
                                 emp_name = name_length,    &
                                 emp_address = address_length

NEXT
```

At the start of program execution, the storage for *badge* is the first 4 bytes of *emp_buffer*, the storage for *social_sec_num* is equal to 9 bytes, and together *name_length* and *address_length* are equal to 2 bytes. The FILL keyword reserves 60 additional bytes of storage. The MAP DYNAMIC statement defines the variables *emp_name* and *emp_address* whose positions and lengths will change at run time. When executed, the REMAP statement defines the FILL area to be equal to *emp_fixed_info* and defines the positions and lengths of *emp_name* and *emp_address*.

When you specify a static string variable, it must be either a variable declared in a MAP or COMMON statement or a parameter declared in a SUB, FUNCTION, PICTURE, or DEF. The actual parameter passed to the procedure must be a static string variable defined in a COMMON, MAP, or RECORD statement.

The following example shows the use of a static string variable as a parameter declared in a SUB. The MAP DYNAMIC statement specifies the input parameter, *input_rec*, as the string to be dynamically defined with the REMAP statement. In addition, the MAP DYNAMIC statement specifies a string array *A* whose elements will point to positions in *input_rec* after the REMAP statement is executed. The REMAP statement defines the length and position of each element contained in array *A*. The FOR...NEXT loop then assigns each element contained in array *A* into array *item*, the target array.

```
SUB deblock (STRING input_rec, STRING item())
 MAP DYNAMIC (input_rec) STRING A(1 TO 3)
 REMAP (input_rec) &
     A(1) = 5, &
     A(2) = 3, &
     A(3) = 4
 FOR I = LBOUND(A) TO UBOUND(A)
   item(I) = A(I)
 NEXT I
END SUB
```

Note that dynamic map variables are local to the program module in which they reside; therefore, REMAP only affects how that module views the buffer.

For more information about using the MAP DYNAMIC and REMAP statements, see the *HP BASIC for OpenVMS Reference Manual*.

8

Creating and Using Data Structures

A **data structure** is a collection of data items that can contain elements or components of different data types.

The RECORD statement lets you create your own data structures. You use the RECORD statement to create a pattern of a data structure, called the **RECORD template**. Once you have created a template, you use it to declare an **instance** of the RECORD, that is, a **RECORD variable**. You declare a RECORD variable just as you declare a variable of any other type: with the DECLARE statement or another declarative statement. A RECORD instance is a variable whose structure matches that of the RECORD template.

The RECORD statement does not create any variables. It only creates a template, or user-defined data type, that you can then use to create variables.

This chapter describes how to create and use data structures.

8.1 RECORD Statement

The RECORD statement names and defines a data structure. Once a data structure (or RECORD) has been named and defined, you can use that RECORD name anywhere that you can use a BASIC data type keyword. You build the data structure using:

- Variables of any valid BASIC data type
- RECORD variables of previously defined RECORD data types
- Any combination of the two

The following example creates a RECORD called *Employee*. *Employee* is a data structure that contains one LONG integer, one 10-character string, one 20-character string, and one 11-character string.

```
RECORD Employee
  LONG Emp_number
  STRING First_name = 10
  STRING Last_name = 20
  STRING Soc_sec_number = 11
END RECORD Empolyee
```

To create instances of this data structure, you use declarative statements. In the following example, the first DECLARE statement creates a variable called *Emp_rec* of data type *Employee*. The second DECLARE statement creates a one-dimensional array called *Emp_array* that contains 1001 instances of the *Employee* data type.

```
DECLARE Employee Emp_rec
DECLARE Employee Emp_array (1000)
```

Any reference to a RECORD component must contain the name of the RECORD instance (that is, the name of the declared variable) and the name of the elementary RECORD component you are accessing, separated by two colons (::). For example, the following program assigns values to an instance of the *Employee* RECORD template:

```
! Record Template

RECORD Employee

  LONG   Emp_number
  STRING First_name = 10
  STRING Last_name  = 20
  STRING Soc_sec_number = 11

END RECORD Employee
! Declarations

DECLARE Employee Emp_rec

DECLARE STRING Social_security

! Program logic starts here.

INPUT 'Employee number'; Emp_rec::Emp_number
INPUT 'First name';      Emp_rec::First_name
INPUT 'Last name';       Emp_rec::Last_name
INPUT 'Social security'; Social_security
IF Social_security <> ""
THEN
   Emp_rec::Soc_sec_number = Social_security
END IF
```

```
PRINT
PRINT "Employee number is: "; Emp_rec::Emp_number
PRINT "First name is: ";      Emp_rec::First_name
PRINT "Last name is: ";       Emp_rec::Last_name
PRINT "Social security is: "; Emp_rec::Soc_sec_number
END
```

When you access an array of RECORD instances, the array subscript should immediately follow the name of the RECORD variable. The following example shows an array of RECORD instances:

```
! Record Template

RECORD Employee

  LONG   Emp_number
  STRING First_name = 10
  STRING Last_name  = 20
  STRING Soc_sec_number = 11

END RECORD
! Declarations

DECLARE Employee Emp_array ( 10 )

DECLARE INTEGER Index

DECLARE STRING Social_security

! Program logic starts here.

FOR Index = 0 TO 10

  PRINT
  INPUT 'Employee number'; Emp_array(Index)::Emp_number
  INPUT 'First name';      Emp_array(Index)::First_name
  INPUT 'Last name';       Emp_array(Index)::Last_name
  INPUT 'Social security'; Social_security
  IF Social_security <> ""
  THEN
     Emp_array(Index)::Soc_sec_number = Social_security
  END IF
NEXT Index

FOR Index = 0 TO 10

PRINT
PRINT "Employee number is: "; Emp_array(Index)::Emp_number
PRINT "First name is: ";      Emp_array(Index)::First_name
PRINT "Last name is: ";       Emp_array(Index)::Last_name
PRINT "Social security is: "; Emp_array(Index)::Soc_sec_number

NEXT Index

END
```

You can have a RECORD that contains an array. When you declare arrays, HP BASIC allows you to specify both lower and upper bounds.

```
RECORD Grade_record

  STRING     Student_name = 30
  INTEGER    Quiz_scores (1 TO 10)    ! Array to hold ten quiz grades.

END RECORD
! Declarations

DECLARE Grade_record Student_grades ( 5 )

!The Student_grades array holds information on six students
!(0 through 5), each of whom has ten quiz grades (1 through 10).

DECLARE INTEGER I,J
!Program logic starts here.

FOR I = 0 TO 5      !This loop executes once for each student.

  PRINT
  INPUT 'Student name'; Student_grades(I)::Student_name

    FOR J = 1 TO 10  !This loop executes ten times for each student.

      PRINT 'Score for quiz number'; J
      INPUT Student_grades(I)::Quiz_scores(J)

    NEXT J
NEXT I

FOR I = 0 TO 5

  PRINT
  PRINT 'Student name: '; Student_grades(I)::Student_name

    FOR J = 1 TO 10

      PRINT 'Score for quiz number'; J; ": ";
      PRINT Student_grades(I)::Quiz_scores(J)

    NEXT J

NEXT I

END
```

Because any reference to a component of a RECORD instance must begin with the name of the RECORD instance, RECORD component names need not be unique in your program. For example, you can have a RECORD component called *First_name* in any number of different RECORD statements. References to this component are unambiguous because every RECORD component reference must specify the record instance in which it resides.

8.1.1 Grouping RECORD Components

A RECORD component can consist of a named group of instances, identified with the keyword GROUP. You use GROUP to refer to a collection of RECORD components, or to create an array of components that have different data types. The GROUP name can be followed by a list of upper and lower bounds, which define an array of the GROUP components. GROUP is valid only within a RECORD block.

The declarations between the GROUP statement and the END GROUP statement are called a GROUP block.

The following example declares a RECORD template of data type *Yacht*. *Yacht* is made up of two groups: *Type_of_yacht* and *Specifications*. Each of these groups is composed of elementary RECORD components. BASIC also allows groups within other groups.

```
RECORD Yacht

  GROUP Type_of_yacht
    STRING Manufacturer = 10
    STRING Model = 10
  END GROUP Type_of_yacht
  GROUP Specifications
    STRING Rig = 6
    STRING Length_over_all = 3
    DECIMAL(5,0) Displacement
    DECIMAL(2,0) Beam
    DECIMAL(7,2) Price
  END GROUP Specifications

END RECORD Yacht
```

8.1.2 RECORD Variants

Sometimes it is useful to have different record components overlay the same record field, in much the same way that multiple maps can overlay the same storage. Such an overlay is called a **RECORD variant**. You use the keywords VARIANT and CASE to set up RECORD variants.

The following example creates a RECORD template for any three kinds of boats:

```
RECORD Boat

  STRING Make  = 10
  STRING Model = 10
  STRING Type_of_boat = 1    ! This field contains the value S, P, or C.
                             ! Value S causes the record instance to be
                             ! interpreted as describing a sailboat, value
                             ! P as describing a powerboat, and value C as
                             ! describing a canoe.
  VARIANT

  CASE     ! Sailboats

    STRING Rig   = 20
  CASE     ! Powerboats

    WORD   Horsepower

  CASE     ! Canoes

    WORD   Length
    WORD   Weight

  END VARIANT

END RECORD
```

The SELECT...CASE statement allows you to access one of several possible RECORD variants in a particular RECORD instance. A RECORD component outside the overlaid fields usually determines which RECORD variant is being used in a particular reference; in this case, the determining RECORD component is *Type_of_boat*. You can use this component in the SELECT expression.

```
! Declarations

DECLARE Boat My_boat

! Main program logic starts here

   .
   .
   .
Input_boat_information:

  INPUT 'Make of boat'; My_boat::Make
  INPUT 'Model';        My_boat::Model
  PRINT 'Type of boat (S = Sailboat, P = Powerboat, C = Canoe)';
  INPUT My_boat::Type_of_boat
  SELECT My_boat::Type_of_boat

  CASE "S"

    INPUT 'Sail rig'; My_boat::Rig

  CASE "P"
```

```
    INPUT 'Horsepower'; My_boat::Horsepower
  CASE "C"

     INPUT 'Length'; My_boat::Length
     INPUT 'Weight'; My_boat::Weight

  CASE ELSE

     PRINT "Invalid type of boat, please try again."

  END SELECT
```

The value of the *Type_of_boat* component determines the format of the variant part of the record.

The following example is a more complex version of the same type of procedure. This program prompts for the RECORD instance components in each variant. When the user responds to the "Wage Class" prompt, the program branches to one of three CASE blocks depending on the value of *Wage_class*.

```
!Record templates

RECORD Emp_wage_class

  STRING Emp_name = 30         ! Employee name string.

  STRING Street = 15           !
  STRING City = 20             ! These components make up the
  STRING State = 2             ! employee address field.
  DECIMAL(5,0) Zip             !

  STRING Wage_class = 1
  VARIANT

    CASE

      GROUP Hourly                     ! Hourly workers.

        DECIMAL(4,2) Hourly_wage       ! Hourly wage rate.
        SINGLE Regular_pay_ytd         ! Regular pay year-to-date.
        SINGLE Overtime_pay_ytd        ! Overtime pay year-to-date.

      END GROUP Hourly
    CASE

      GROUP Salaried                   ! Salaried workers.

        DECIMAL(7,2) Yearly_salary     ! Yearly salary.
        SINGLE Pay_ytd                 ! Pay year-to-date.

      END GROUP Salaried
    CASE

      GROUP Executive                  ! Executives.

        DECIMAL(8,2) Yearly_salary     ! Yearly salary.
        SINGLE Pay_ytd                 ! Pay year-to-date.
        SINGLE Expenses_ytd            ! Expenses year-to-date.
```

```
      END GROUP Executive

  END VARIANT
END RECORD
! Declarations:

  DECLARE Emp_wage_class Emp

! Main Program logic starts here.

LINPUT "Name"; Emp::Emp_name          ! Use LINPUT statements for
LINPUT "Street"; Emp::Street          ! string fields so the entire
                                      ! string is assigned to the
LINPUT "State"; Emp::State            ! variable.
INPUT  "Zip Code"; Emp::Zip
LINPUT "Wage Class"; Emp::Wage_class
SELECT Emp::Wage_class

CASE "A"
  INPUT 'Rate';Emp::Hourly_wage
  INPUT 'Regular pay';Emp::Regular_pay_ytd
  INPUT 'Overtime pay';Emp::Overtime_pay_ytd

CASE "B"
  INPUT 'Salary';Emp::Salaried::yearly_salary
  INPUT 'Pay YTD';Emp::Salaried::pay_ytd

CASE "C"
  INPUT 'Salary';Emp::Executive::yearly_salary
  INPUT 'Pay YTD';Emp::Executive::pay_ytd
  INPUT 'Expenses';Emp::Expenses_ytd

END SELECT
```

Variant fields can appear anywhere within the RECORD instance. When you use RECORD variants, you imply that any RECORD instance can contain any one of the listed variants. Therefore, if each variant requires a different amount of space, BASIC uses the case that requires the most storage to determine the space allocated for each RECORD instance.

8.1.3 Accessing RECORD Components

To access a particular elementary component within a RECORD that contains other groups, you use the name of the declared RECORD instance, the group name (or group names, if groups are nested), and the elementary component name, each separated by double colons (::).

In the following example, the PRINT statement displays the *Rig* component in the *Specifications* group in the variable named *My_yacht*. The RECORD instance name qualifies the group name and the group name qualifies the elementary RECORD component. The elementary component name, qualified by all intermediate group names and by the RECORD instance name, is called

a **fully qualified component**. The full qualification of a component is called a **component path name**.

```
DECLARE Yacht My_yacht

   .
   .
   .

PRINT My_yacht::Specifications::Rig
```

Because it is cumbersome to specify the entire component path name, BASIC allows **elliptical references** to RECORD components. GROUP names are optional in the component path name unless:

- A RECORD contains more than one component with the same name
- The GROUP is an array

The rules for using elliptical references are as follows:

- You must always specify the RECORD instance, that is, the name of the declared variable.
- You must always specify any dimensioned group.
- You may omit any other intermediate component names.
- You must specify the final component name.

The following example shows that using the complete component path name is valid but not required. The assignment statement uses the fully qualified component name; the PRINT statement uses an elliptical reference to the same component, omitting *Extended_family* and *Nuclear_family* GROUP names. Note that the *Children* GROUP name *is* required because the GROUP is an array; the elliptical reference to this component must include the desired array element, in this case the second element of the *Children* array.

```
! RECORD templates:

RECORD Family

  GROUP Extended_family

    STRING Grandfather(1) = 30    ! Two-element fixed-length string
    STRING Grandmother(1) = 30    ! arrays for the names of maternal
                                  ! and paternal grandparents.
    GROUP Nuclear_family

      STRING Father = 30          ! Fixed-length strings for the names
      STRING Mother = 30          ! of parents.
```

```
        GROUP Children (10)       ! An 11-element array for the names and
                                  ! gender of children.
          STRING Kid = 10
          STRING Gender = 1

        END GROUP Children

    END GROUP Nuclear_family

  END GROUP Extended_family

END RECORD
! Declarations

DECLARE Family My_family
! Program logic starts here.

My_family::Extended_family::Nuclear_family::Children(1)::Kid = "Johnny"

PRINT My_family::Children(1)::Kid

END
```

Output

```
Johnny
```

```
! RECORD Templates.

RECORD Test

  INTEGER Test_integers(2)     ! 3-element array of integers.

  GROUP Group_1                ! Single GROUP containing:

    REAL   My_number           ! a real number and
    STRING Group_1_string      ! a 16-character (default) string

  END GROUP
  GROUP Group_2(5)             ! A 6-element GROUP, each element containing:

    INTEGER My_number          ! an integer and
    DECIMAL Group_2_decimal    ! a DECIMAL number.

  END GROUP

END RECORD
! Declarations

DECLARE Test Array_of_test(10)  ! Create an 11-element array of type Test...
DECLARE Test Single_test        ! ...and a separate single instance of type
                                ! Test.
```

The minimal reference to the string *Group_1_string* in RECORD instance *Array_of_test* is as follows:

```
Array_of_test(i)::Group_1_string
```

In this case, *i* is the subscript for array *Array_of_test*. Because the RECORD instance is itself an array, the reference must include a specific array element.

Because *Single_test* is not an array, the minimal reference to string *Group_1_string* in RECORD instance *Single_test* is as follows:

```
Single_test::Group_1_string
```

The minimal reference for the REAL variable *My_number* in GROUP *Group_1* in RECORD instance *Array_of_test* is as follows:

```
Array_of_test(i)::Group_1::My_number
```

Here, *i* is the subscript for array *Array_of_test*. The minimal reference to the REAL variable *My_number* in RECORD instance *Single_test* is as follows:

```
Single_test::Group_1::My_number
```

Because there is a variable named *My_number* in groups *Group_1* and *Group_2*, you must specify either *Group_1::My_number* or *Group_2(i)::My_number*. In this case, extra component names are required to resolve an otherwise ambiguous reference.

The minimal reference to the DECIMAL variable *Group_2_decimal* in RECORD instances *Array_of_test* and *Single_test* are the fully qualified references. In the following examples, *i* is the subscript for array *Array_of_test* and *j* is an index into the group array *Group_2*. Even though *Group_2_decimal* is a unique component name within RECORD instance *Single_test*, the element of array *Group_2* must be specified. In this case, the extra components must be specified because each element of GROUP *Group_2* contains a component named *Group_2_decimal*.

```
Array_of_test(i)::Group_2(j)::Group_2_decimal
```

```
Single_test::Group_2(j)::Group_2_decimal
```

You can assign all the values from one RECORD instance to another RECORD instance, as long as the RECORD instances are identical except for names.

In the following example, RECORD instances *First_test1*, *Second_test1*, and the individual elements of array *Array_of_test1* have the same form: an array of four groups, each of which contains a 10-byte string variable, followed by a REAL variable, followed by an INTEGER variable. Any of these RECORD instances can be assigned to one another.

```
!RECORD Templates

RECORD Test1

  GROUP Group_1(4)

    STRING   My_string_1 = 10
    REAL     My_real_1
    INTEGER  My_integer_1
```

```
  END GROUP
END RECORD
RECORD Test2
  GROUP Group_2
    STRING  My_string_2 = 10
    REAL    My_real_2
    INTEGER My_integer_2
  END GROUP
END RECORD
RECORD Test3
  STRING   My_string_3 = 10
  REAL     My_real_3
  INTEGER  My_integer_3
END RECORD
!Declarations
DECLARE Test1 First_test1,         &
              Second_test1,        &
              Array_of_test1(3)
DECLARE Test2 First_test2
DECLARE Test3 First_test3,         &
              Array_of_test3(10)
!Program logic starts here
! A single RECORD instance is assigned to another single instance
First_test1 = Second_test1
! An array element is assigned to a single instance
Second_test1 = Array_of_test1(2)
! And vice versa
Array_of_test1(2) = Second_test1
```

Further, you can assign values from single RECORD instances to groups contained in other instances.

In the following example, *Array_of_test1* and *First_test1* do not have the same form because *Array_of_test1* is an array of RECORD *Test1* and *First_test1* is a single instance of RECORD *Test1*. Therefore, *First_test1* and *Array_of_test1* cannot be assigned to one another.

```
! A single instance is assigned to one group

Array_of_test1(3)::Group_1(2) = First_test1

! An array element is assigned a value from
! a group contained in another array instance
Array_of_test3(5) = Array_of_test1(3)::Group_1(3)
```

The examples shown in this chapter explain the mechanics of using data structures. See Chapter 12 for more information about using data structures as parameters. See Chapter 13 for more information about using data structures for file input and output.

9
Program Control

This chapter describes the HP BASIC control statements.

HP BASIC normally executes statements sequentially. Control statements let you change this sequence of execution. HP BASIC control statements can alter the sequence of program execution at several levels:

- Statement modifiers control the execution of a single statement.
- Loops or decision blocks control the execution of a block of statements.
- Branching statements such as GOTO and ON GOTO pass control to statements or local subroutines.
- The EXIT and ITERATE statements explicitly control loops or decision blocks.
- The SLEEP, WAIT, STOP and END control statements suspend or halt the execution of your entire program.

9.1 Statement Modifiers

Statement modifiers are control structures that operate on a single statement. Statement modifiers let you execute a statement conditionally or create a loop. The following are BASIC 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 cannot be modified.

9.1.1 IF Modifier

The IF modifier tests a conditional expression. If the conditional expression is true, HP BASIC executes the statement. If it is false, HP BASIC does not execute the modified statement but continues execution at the next program statement. The following is an example of a statement using the IF modifier:

```
PRINT A IF (A < 5)
```

9.1.2 UNLESS Modifier

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

```
PRINT A UNLESS (A < 5)
```

This is equivalent to the following:

```
PRINT A IF A >= 5
```

9.1.3 FOR Modifier

The FOR modifier creates a loop on a single line. The following is an example of a loop created using the FOR modifier:

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

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

```
B = B + 1 UNTIL (A  - B) < 0.0001
```

9.1.5 WHILE Modifier

The WHILE modifier repeats a statement as long as a conditional expression is true. Like the UNTIL and FOR modifiers, the WHILE modifier lets you create single-line loops. In the following example, HP BASIC replaces the value of *A* with *A*/2, as long as the absolute value of *A* is greater than one-tenth. Note that you can inadvertently create an infinite loop if the terminating condition is never reached.

```
A = A / 2 WHILE ABS(A) > 0.1
```

9.1.6 Nesting Modifiers

If you append more than one modifier to a statement, you are **nesting modifiers**. HP 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.

In the following example, HP BASIC first tests the rightmost modifier of the first PRINT statement. Because this condition is false, HP BASIC executes the following PRINT statement and tests the rightmost modifier. Because this condition is met, HP BASIC tests the leftmost modifier of the same PRINT statement. This condition, however, is not met. Therefore, HP BASIC executes the following PRINT statement. Because both conditions are met in the third PRINT statement, HP BASIC prints the value of *C*.

```
A = 5
B = 10
C = 15

PRINT "A =";A IF A = 5 UNLESS C = 15
PRINT "B =";B UNLESS C = 15 IF B = 10
PRINT "C =";C IF B = 10 UNLESS C = 5
END
```

Output

```
C = 15
```

9.2 Loops

Loops allow you to repeat the execution of a set of statements. This set of statements is called a **loop block**. There are three types of HP BASIC program loops:

FOR...NEXT
WHILE...NEXT
UNTIL...NEXT

Note that these types of loops can be nested, that is, lexically located one inside another.

9.2.1 FOR...NEXT Loops

In a FOR...NEXT loop, you specify a loop control variable (the loop index) that determines the number of loop iterations. This number must be a scalar (unsubscripted) variable. When HP BASIC begins execution of a FOR...NEXT loop, the starting and ending values of the loop control variable are known.

The FOR statement assigns the control variable a starting value and an ending value. You can use the optional STEP clause to specify the amount to be added to the loop control variable after each loop iteration.

When a FOR loop block executes, the HP BASIC compiler performs the following steps:

1. Evaluates the starting value and assigns it to the control variable.
2. Evaluates the ending value and the step value and assigns these results to temporary storage locations.
3. Tests whether the ending value has been exceeded. If the ending value has already been exceeded, HP BASIC executes the statement following the NEXT statement. If the ending value has not been exceeded, HP BASIC executes the statements in the loop.
4. Adds the step value to the control variable and transfers control to the FOR statement, which tests whether the ending value has been exceeded. Steps 3 and 4 are repeated until the ending value is exceeded.

Note that HP BASIC performs the test before the loop executes. When the control variable exceeds the ending value, HP BASIC exits the loop, and then subtracts the step value from the control variable. This means that after loop execution, the value of the control variable is the value last used in the loop, not the value that caused loop termination.

Example 9–1 assigns the values 1 to 10 to consecutive array elements 1 to 10 of *New_array*, and Example 9–2 assigns consecutive multiples of 2 to the odd-numbered elements of *New_array*.

Example 9–1 Assigning Values to Consecutive Array Elements

```
FOR I% = 1% TO 10%
    New_array(I%) = I%
NEXT I%
```

Example 9–2 Assigning Consecutive Multiples to Odd-Numbered Elements of Array

```
FOR I% = 1% TO 10% STEP 2
    New_array(I%) = I% + 1%
NEXT I%
```

Note that the starting, ending, and step values can be run-time expressions. You can have HP BASIC calculate these values when the program runs, as opposed to using a constant value. For instance, the following example assigns sales information to array *Sales_data*. The number of iterations depends on the value of the variable *Days_in_month*, which represents the number of days in that particular month.

```
FOR I% = 1% TO Days_in_month
    Sales_data(I%) = Quantity_sold
NEXT I%
```

Because the starting, ending, and step values can be numeric expressions, they are not evaluated until the program runs. This means that you can have a FOR...NEXT loop that does not execute. The following example prompts the user for the starting, ending, and step values for a loop, and then tries to execute that loop. The loop executes zero times because it is impossible to go from 0 to 5 using a step value of -1.

```
counter% = 0%

INPUT "Start"; start%
INPUT "Finish"; finish%
INPUT "Step value"; step_val%

FOR I% = start% TO finish% STEP step_val%
    counter% = counter% + 1%
NEXT I%

PRINT "This loop executed"; counter%; "times."
```

Output

```
Start? 0
Finish? 5
Step value? -1
This loop executed 0 times.
```

Whenever possible, you should use integer variables to control the execution of FOR...NEXT loops 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 imprecision.

In the following example, the first loop uses an integer control variable while the second uses a floating-point control variable. The first loop executes 100 times and the second 99 times. After the ninety-ninth iteration of the second loop, the internal representation of the value of *Floating_point_variable* exceeds 10 and BASIC exits the loop. Because the first loop uses integer values to control execution, HP BASIC does not exit the loop until *Integer_variable* equals 100.

```
Loop_count_1 = 0%
Loop_count_2 = 0%

FOR Integer_variable = 1% to 100% STEP 1%
    Loop_count_1 = Loop_count_1 + 1%
NEXT Integer_variable

FOR Floating_point_variable = 0.1 to 10 STEP 0.1
    Loop_count_2 = Loop_count_2 + 1%
NEXT Floating_point_variable

PRINT "Integer loop count:"; Loop_count_1
PRINT "Integer loop end  :"; Integer_variable
PRINT "Real loop count:   "; Loop_count_2
PRINT "Real loop end:     "; Floating_point_variable
```

Output

```
Integer loop count:  100
Integer loop end:    100
Real loop count:     99
Real loop end:       9.9
```

Although it is not recommended programming practice, you can assign a value to a FOR...NEXT loop's control variable while in the loop. This affects the number of times a loop executes. For example, assigning a value that exceeds the ending value of a loop will cause the loop's execution to end as soon as HP BASIC performs the termination test in the FOR statement. Assigning values to ending or step variables, however, has no effect at all on the loop's execution.

9.2.2 WHILE...NEXT Loops

A WHILE...NEXT statement uses a conditional expression to control loop execution; the loop is executed as long as a given condition is true. A WHILE...NEXT loop is useful when you do not know how many loop iterations are required.

In the following example, the first statement instructs the user to input data and then type DONE when finished. After the user enters the first piece of input, HP BASIC executes the WHILE...NEXT loop. If the first input value is not "DONE", the loop executes and prompts the user for another input value.

Once the user enters this input value, the WHILE...NEXT loop once again checks to see if this value corresponds to “DONE”. The loop will continue executing until the user types “DONE” in response to the prompt.

```
INPUT 'Type "DONE" when finished'; Answer

WHILE (Answer <> "DONE")
   .
   .
   .
      INPUT "More data"; Answer
NEXT
```

Note that the NEXT statement in the WHILE...NEXT and UNTIL...NEXT loops does not increment a control variable; your program must change a variable in the conditional expression or the loop will execute indefinitely.

The evaluation of the conditional expression determines whether the loop executes. The test is performed (that is, the conditional expression is evaluated) before the first iteration; if the value is false (0), the loop does not execute.

It can be useful to intentionally create an infinite loop by coding a WHILE...NEXT loop whose conditional expression is always true. When doing this you must take care to provide a way out of the loop. You can do this with an EXIT statement or by trapping a run-time error. See Chapter 15 for more information about trapping run-time errors.

9.2.3 UNTIL...NEXT Loops

The UNTIL...NEXT loop performs like a WHILE...NEXT loop, except that the logical sense of the conditional expression is reversed; that is, the UNTIL...NEXT loop executes until a given condition is true.

An UNTIL...NEXT loop executes repeatedly for as long as the conditional expression is false. Note that in UNTIL...NEXT loops, the NEXT statement does not increment a control variable. You must explicitly change a variable in the conditional expression or the loop will execute indefinitely.

It is possible to code the WHILE...NEXT loop with a UNTIL...NEXT loop, as shown in the following example. These loops are equivalent except for the logical sense of the termination test (WHILE *Answer* <> “DONE” as opposed to UNTIL *Answer* = “DONE”).

```
INPUT 'Type "DONE" when finished.'; Answer
UNTIL (Answer = "DONE")
   .
   .
   .
      INPUT "More data"; Answer
NEXT
```

9.2.4 Nesting Loops

When a loop block is entirely contained in another loop block, it is called a **nested loop**.

The following example declares a two-dimensional array and uses nested FOR...NEXT loops to fill the array elements with sales information. The inner loop executes 16 times for each iteration of the outer loop. This example assigns a value to each of the 256 elements of the array.

```
DECLARE
   INTEGER
      Column_number,
      Row_number
   REAL
      Sales_info,
      Two_dim_array (15%, 15%)
FOR Row_number = 0% TO 15%
    FOR Column_number = 0% to 15%
        INPUT "Please enter the sales information";Sales_info
        Two_dim_array (Row_number, Column_number) = Sales_info
    NEXT Column_number
NEXT Row_number
```

Note that in nested loops the inner loop is entirely contained in the outer loop; nested loops cannot overlap.

9.3 Unconditional Branching (GOTO Statement)

The GOTO statement specifies which program line the HP BASIC compiler is to execute next, regardless of that line's position in the program. If the statement at the target line number or label is nonexecutable (such as an REM statement), HP BASIC transfers control to the next executable statement following the target line number.

You can use a GOTO statement to exit from a loop; however, it is better programming practice to use the EXIT statement.

9.4 Conditional Branching

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

- ON...GOTO...OTHERWISE
- IF...THEN...ELSE
- SELECT...CASE

9.4.1 ON...GOTO...OTHERWISE Statement

The ON...GOTO...OTHERWISE statement tests the value specified after the ON keyword. If the value is 1, HP BASIC transfers control to the first target in the list; if the value is 2, control passes to the second target, and so on. If the value is less than 1 or greater than the number of targets in the list, HP BASIC transfers control to the target specified in the OTHERWISE clause. For example:

```
Menu:
  PRINT "Would you like to change:"
  PRINT "1.  First name"
  PRINT "2.  Last name"

INPUT CHOICE%

ON CHOICE% GOTO First_name, Last_name OTHERWISE Other_choice

First_name:
   INPUT "First name"; firstname$
   GOTO Done

Last_name:
   INPUT "Last name"; lastname$
   GOTO Done

Other_choice:
    PRINT "Invalid choice"
    PRINT "Let's try again"
    GOTO Menu

Done:
    END
```

Note that if you do not supply an OTHERWISE clause and the control variable is less than 1 or greater than the number of targets, BASIC signals "ON statement out of range (ERR = 58)".

9.4.2 IF...THEN...ELSE Statement

The IF...THEN...ELSE statement evaluates a conditional expression and uses the result to determine which block of statements to execute next. If the conditional expression is true, HP BASIC executes the statements in the THEN clause. If the conditional expression is false, HP BASIC executes the statements in the ELSE clause, if one is present. If the conditional expression is false and there is no ELSE clause, HP BASIC executes the statement immediately following the END IF statement.

In the following example, HP BASIC evaluates the conditional expression *number* < 0. If the input value of *number* is less than zero, the conditional expression is true. HP BASIC then executes the statements in the THEN clause, skips the statement in the ELSE clause, and transfers control to the statement following the END IF. If the value of *number* is greater than or equal to zero, the conditional expression is false. HP BASIC then skips the statements in the THEN clause and executes the statement in the ELSE clause.

```
INPUT "Input number"; number

IF (number < 0)
THEN
   number = - number
   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
END
```

Output

```
Input number? -9
That square root is imaginary
The square root of its absolute value is 3
```

Do not neglect to end an IF...THEN...ELSE statement. After an IF block is executed, control is transferred to the statement immediately following the END IF. If there is no END IF, HP BASIC transfers control to the next line number. Code between the keyword ELSE and the next line number becomes part of the ELSE clause. If there are no line numbers, the HP BASIC compiler ignores the remaining program code from the keyword ELSE to the end of the program. Therefore, it is important to end IF statements.

IF...THEN...ELSE statements can be nested. In an inner nesting level, if an END IF is not present, the BASIC compiler treats the presence of an ELSE clause for an IF statement in an outer nesting level as an implicit END IF for all unterminated IF statements at that point. For example, in the following construction, the third ELSE terminates both inner IFs:

```
IF expression
THEN
    IF expression
    THEN
        statement-list
    ELSE
        IF expression
        THEN
            statement-list
        ELSE
            statement-list
ELSE
```

In the following example, the first IF...THEN...ELSE statement is ended by END IF, and works as expected. Because the second IF...THEN...ELSE statement is not terminated by END IF, the HP BASIC compiler assumes that the last PRINT statement in the program is part of the second ELSE clause.

```
10  DECLARE INTEGER light_bulb
    DECLARE INTEGER circuit_switch
    DECLARE INTEGER CONSTANT Opened  = 0
    DECLARE INTEGER CONSTANT Closed =  1

    PRINT "Please enter zero or one, corresponding to the circuit"
    PRINT "switch being open or closed"
    INPUT On_off_val
❶     IF On_off_val = Opened
      THEN
        PRINT "The light bulb is off."
      ELSE
        PRINT "The light bulb is on."
    END IF
    IF On_off_val = Closed
      THEN
        PRINT "The light bulb is on."
      ELSE
        PRINT "The light bulb is off."
❷     PRINT "That's all for now."
20  END
```

❶ When you run the program, the first IF...THEN...ELSE statement will always execute correctly.

❷ The final PRINT statement will execute only when the value of *On_off_val* is 1 (that is, *closed*), because the compiler considers this PRINT statement to be part of the second ELSE clause.

Output 1

```
Please enter zero or one, corresponding to the circuit
switch being open or closed
? 0
The light bulb is off.
The light bulb is off.
That's all for now.
```

Output 2

```
Please enter zero or one, corresponding to the circuit
Switch being open or closed
? 1
The light bulb is on.
The light bulb is on.
```

Note that a statement in a THEN or ELSE clause can be followed by a modifier. In this case, the modifying IF applies only to the statement that immediately precedes it.

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

9.4.3 SELECT...CASE Statement

The SELECT...CASE statement lets you specify an expression (the SELECT expression), any number of possible values (cases) for the SELECT expression, and a list of statements (a CASE block) for each case. The SELECT expression can be a numeric or string value. CASE values can be single or multiple values, one or more ranges of values, or relationships. When a match is found between the SELECT expression and a CASE value, the statements in the following CASE block are executed. Control is then transferred to the statement following the END SELECT statement.

In the following example, the CASE values appear to overlap; that is, the CASE value that tests for values greater than or equal to 0.5 also includes the values greater than or equal to 1.0. However, HP BASIC executes the statements associated with the *first* matching CASE statement and then transfers control to the statement following END SELECT. In this program, each range of values is tested *before* it overlaps in the next range. Because the

compiler executes the first matching CASE statement, the overlapping values do not matter.

```
DECLARE REAL Stock_change

INPUT "Please enter stock price change";Stock_change

SELECT Stock_change

CASE <= 0.5
     PRINT "Don't sell yet."

CASE <= 1.0
     PRINT "Sell today."

CASE ELSE
     PRINT "Sell NOW!"

END SELECT
END
```

Output

```
Please enter stock price change? 2.1
Sell NOW!
```

If no match is found for any of the specified cases and there is no CASE ELSE block, HP BASIC transfers control to the statement following END SELECT without executing any of the statements in the SELECT block.

SELECT...CASE lets you use run-time expressions for both SELECT expressions and CASE values. The following example uses HP BASIC built-in string functions to examine command input:

```
! This program is a skeleton command processor.
! It recognizes three VAX BASIC Environment commands:
!
!      SAVE
!      SCRATCH
!      OLD

DECLARE INTEGER CONSTANT True  = -1
DECLARE INTEGER CONSTANT False =  0

DECLARE STRING CONSTANT Null_input = ""   !This is the null string.

DECLARE STRING Command

! Main program logic starts here.

Command_loop:

WHILE True          ! This loop executes until the user types only a
                    ! carriage return in response to the prompt.
```

```
    PRINT
    PRINT "Please enter a command (uppercase only)."
    PRINT "Type a carriage return when finished."
    INPUT Command
    PRINT

    SELECT Command

    CASE Null_input                     ! If user types RETURN,
                                        ! exit from the loop
      GOTO Done                         ! and end the program.

    ! The next three cases use the SEG$ and LEN string functions.
    ! LEN returns the length of the typed string, and SEG$ searches
    ! the string literals ("SAVE", "SCRATCH", and "OLD") for a
    ! match up to that length.  Note that if the user types an "S",
    ! it is interpreted as a SAVE command only because SAVE is the
    ! first case tested.

    CASE SEG$ ( "SAVE", 1%, LEN (Command) )
      PRINT "That was a SAVE command."

    CASE SEG$ ( "SCRATCH", 1%, LEN (Command) )
      PRINT "That was a SCRATCH command."

    CASE SEG$( "OLD", 1%, LEN (Command) )
      PRINT "That was an OLD command."

    CASE ELSE
      PRINT "Invalid command, please try again."

    END SELECT
  NEXT

Done:
     END
```

9.5 EXIT and ITERATE Statements

This section describes the EXIT and ITERATE statements and shows their use with nested control structures.

The ITERATE and EXIT statements let you explicitly control loop execution. These statements can be used to transfer control to the top or bottom of a control structure.

You can use EXIT to transfer control out of any of these structures:

- FOR...NEXT loops
- WHILE...NEXT loops
- UNTIL...NEXT loops
- IF...THEN...ELSE blocks

- SELECT...CASE blocks
- SUB, FUNCTION, and PICTURE subprograms
- DEF functions, and programs

In the case of control structures, EXIT passes control to the first statement following the end of the control structure.

You can use ITERATE to explicitly reexecute a FOR...NEXT, WHILE...NEXT, or UNTIL...NEXT loop. EXIT and ITERATE statements can appear only within the code blocks you want to leave or reexecute.

Executing the ITERATE statement is equivalent to transferring control to the loop's NEXT statement. The termination test is still performed when the NEXT statement transfers control to the top of the loop. In addition, transferring control to the NEXT statement means that a FOR loop's control variable is incremented by the STEP value.

Supplying a label for every loop lets you state explicitly which loop to leave or reexecute. If you do not supply a label for the ITERATE statement, HP BASIC reexecutes the innermost active loop. For example, if an ITERATE statement (that does not specify a label) is executed in the innermost of three nested loops, only the innermost loop is reexecuted.

In contrast, labeling each loop and supplying a label argument to the ITERATE statement lets you reexecute any of the loops. A label name also helps document your code. Because you must use a label with EXIT and it is sometimes necessary to use a label with ITERATE, you should always label the structures you want to control with these statements.

The following example shows the use of both the EXIT and ITERATE statements. 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.

```
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
END
```

9.6 Executing Local Subroutines

In HP BASIC, a **subroutine** is a block of code accessed by a GOSUB or ON GOSUB statement. It must be 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 HP BASIC statement, including a REM statement. You do not have to transfer control to the first line of the subroutine. Instead, you can include several entry points into the same subroutine. You can also reference subroutines by using a GOSUB or ON GOSUB statement to another subroutine.

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

9.6.1 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 conditionally, depending on a specified condition.

The following example first assigns a value of 5 to the variable *A*, then transfers control to the subroutine labeled *Times_two*. This subroutine replaces the value of *A* with *A* multiplied by 2. The subroutine's RETURN statement transfers control to the first PRINT statement, which displays the changed value. The program calls the subroutine two more times, with different values for *A*. Each time, the RETURN transfers control to the statement immediately following the corresponding GOSUB.

```
A = 5
GOSUB Times_two
PRINT A

A = 15
GOSUB Times_two
PRINT A

A = 25
GOSUB Times_two
PRINT A

GOTO Done

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

Done:
    END
```

Output

```
10
30
50
```

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

9.6.2 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 that you can return control conditionally, depending on a specified condition.

HP BASIC tests the value of the integer expression. If the value is 1, control transfers to the first line number or label in the list; if the value is 2, control passes to the second line number or label, and so on. If the control variable's value is less than 1 or greater than the number of targets in the list, HP 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 1 or greater than the number of targets, BASIC signals "ON statement out of range (ERR=58)". For example:

```
INPUT "Please enter first integer value"; First_value%
INPUT "Please enter second integer value"; Second_value%

Choice:
    PRINT "Do you want to perform:"
    PRINT "1.  Multiplication"
    PRINT "2.  Division"
    PRINT "3.  Exponentiation"

INPUT Selection%

ON Selection% GOSUB Mult, Div, Expon OTHERWISE Wrong
GOTO Done

Mult:
    Result% = First_value% * Second_value%
    PRINT Result%
    RETURN

Div:
    Result% = First_value / Second_value%
    PRINT Result%
    RETURN

Expon:
    Result% = First_value% ** Second_value%
    PRINT Result%
    RETURN

Wrong:
    PRINT "Invalid selection"
    RETURN

Done:
    END
```

9.7 Suspending and Halting Program Execution

The following HP BASIC statements suspend program execution:

SLEEP
WAIT

These statements cause HP BASIC either to suspend program execution for a specified time or to wait a certain period of time for user input.

After execution of the last statement, a HP BASIC program automatically halts and closes all files. However, you can explicitly halt program execution by using one of the following statements:

STOP
END

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.

9.7.1 SLEEP Statement

The SLEEP statement suspends program execution for a specified number of seconds. The following program waits two minutes (120 seconds) after receiving the input string, and then prints it:

```
INPUT "Type a string of characters"; C$
SLEEP 120%
PRINT C$
END
```

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 the condition at specified intervals.

9.7.2 WAIT Statement

You use the WAIT statement only with terminal input statements such as INPUT, INPUT LINE, and LINPUT. For example, the following program prompts for input, then waits 30 seconds for your response. If the program does not receive input in the specified time, HP BASIC signals “Keyboard wait exhausted (ERR=15)” and exits the program.

```
WAIT 30%
INPUT "You have 30 seconds to type your password"; PSW$
END
```

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

In the following example, the first WAIT statement causes the first INPUT statement to wait 30 seconds for a response. The WAIT 0% statement disables this 30-second requirement for all subsequent INPUT statements.

```
WAIT 30%
INPUT "You have 30 seconds to type your  password"; PSW$
WAIT 0%
INPUT "What directory do you want to go to"; DIR$
```

9.7.3 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.

When HP BASIC executes a STOP statement, it signals "STOP at line <line-num>."

If you compile, link, and execute a program containing a STOP statement, HP BASIC displays a number sign (#) prompt when the STOP statement is encountered. At this point, you can enter:

- CONTINUE (to continue program execution)
- EXIT (to return to DCL command level)

9.7.4 END Statement

The END statement marks the end of a main program. When HP BASIC executes an END statement, it closes all files and halts program execution.

The END statement is optional in HP BASIC programs. However, it is good programming practice to include it. The END statement must be the last statement in the main program.

The END statement returns you to DCL command level.

10
Functions

A **function** is a single statement or group of statements that perform operations on operands and return the result to your program. HP BASIC has built-in functions that perform numeric and string operations, conversions, and date and time operations. This chapter describes only a selected group of built-in functions. For a complete description of all HP BASIC built-in functions, see the *HP BASIC for OpenVMS Reference Manual*.

This chapter also describes user-defined functions. HP 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. For more information about creating external functions with the FUNCTION statement, see Chapter 12.

Once you create and declare a function, you can invoke it like a built-in function.

10.1 Built-In Functions

The functions described in this section let you perform sophisticated manipulations of string and numeric data. HP BASIC also provides algebraic, exponential, trigonometric, and randomizing mathematical functions.

10.1.1 Numeric Functions

Numeric functions generally return a result of the same data type as the function's parameter. For example, if you pass a DOUBLE argument to any of the trigonometric functions, they return a DOUBLE result.

If the format of a HP BASIC function specifies an argument of a particular data type, HP BASIC converts the actual argument supplied to the specified data type. For example, if you supply an integer argument to a function that expects a floating-point number, HP BASIC converts the argument to a floating-point number. Floating-point arguments that are passed to integer functions are truncated, not rounded.

The following sections discuss the HP BASIC built-in numeric functions.

10.1.1.1 ABS Function

The ABS function returns a floating-point number that equals the absolute value of a specified numeric expression. For example:

```
READ A,B
DATA 10,-35.3
NEW_A = ABS(A)
PRINT NEW_A; ABS(B)
END
```

Output

```
 10  35.3
```

The ABS function always returns a number of the default floating-point data type.

10.1.1.2 INT and FIX Functions

The INT function returns the floating-point value of the largest integer less than or equal to a specified expression. The INT function 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. FIX always returns a number of the default floating-point type.

The following example shows the differences between the INT and FIX functions. Note that the value returned by FIX(-45.3) differs from the value returned by INT(-45.3).

```
PRINT INT(23.553); FIX(23.553)
PRINT INT(3.1); FIX(3.1)
PRINT INT(-45.3); FIX(-45.3)
PRINT INT(-11); FIX(-11)
END
```

Output

```
 23  23
 3  3
-46 -45
-11 -11
```

10.1.1.3 SIN, COS, and TAN Functions

The SIN, COS, and TAN functions return the sine, cosine, and tangents of an angle in radians or degrees, depending on which angle clause you choose with the OPTION statement. If you supply a floating-point argument to the SIN, COS, and TAN functions, they return a number of the same floating-point type. 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.

The following example accepts an angle in degrees, converts the angle to radians, and prints the angle's sine, cosine, and tangent:

```
!CONVERT ANGLE (X) TO RADIANS, AND
!FIND SIN, COS AND TAN
PRINT "DEGREES", "RADIANS", "SINE", "COSINE","TANGENT"
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%

DATA 0,10,20,30,360,45
END
```

Output

```
DEGREES       RADIANS       SINE          COSINE        TANGENT

 0             0             0             1             0

 10            .174533       .173648       .984808       .176327

 20            .349066       .34202        .939693       .36397

 30            .523599       .5            .866025       .57735

 360           6.28319       .174846E-06   1             .174846E-06

 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, HP BASIC signals "Division by 0" (ERR=61).

10.1.1.4 SQR Function

The SQR function returns the square root of a number. For example:

```
PRINT SQR (2)
```

Output

```
1.41421
```

10.1.1.5 LOG10 Function

A **logarithm** is the exponent of some number (called a base). Common logarithms use the base 10. The common logarithm of a number *n*, for example, 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. The following example calculates the common logarithms of all multiples of 10 from 10 to 100, inclusively:

```
FOR I% = 10% TO 100% STEP 10%
    PRINT LOG10(I%)
NEXT I%
END
```

Output

```
 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. If you supply an integer argument, LOG10 converts it to the default floating-point data type and returns a value of that type.

10.1.1.6 EXP Function

The EXP function returns the value of *e* raised to a specified power. The following example prints the value of *e* and *e* raised to the second power:

```
READ A,B
DATA 1,2
PRINT 'e RAISED TO THE POWER'; A; " EQUALS"; EXP(A)
PRINT 'e RAISED TO THE POWER'; B; " EQUALS"; EXP(B)
END
```

Output

```
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. If you supply an integer argument, EXP converts it to the default floating-point data type and returns a value of that type.

10.1.1.7 RND Function

The RND function returns a number greater than or equal to zero and less than 1. 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.

```
PRINT RND,RND,RND,RND
END
```

Output 1

```
 .76308      .179978      .902878      .88984
```

Output 2

```
 .76308      .179978      .902878      .88984
```

With the RANDOMIZE statement, you can change the RND function's starting condition and generate 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.

```
RANDOMIZE
PRINT RND,RND,RND,RND
END
```

Output 1

```
 .403732       .34971       .15302       .92462
```

Output 2

```
 .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 following formula:

```
(B-A)*RND + A
```

The following program produces 10 numbers in the open range 4 to 6:

```
FOR I% = 1% TO 10%
    PRINT (6%-4%) * RND + 4
NEXT I%
END
```

Output

```
 5.52616
 4.35996
 5.80576
 5.77968
 4.77402
 4.95189
 5.76439
 4.37156
 5.2776
 4.53843
```

10.1.2 Data Conversion Functions

HP BASIC provides built-in functions that can perform the following:

- Convert a 1-character string to the character's ASCII value and vice versa
- Translate strings from one data format to another, for example, EBCDIC to ASCII

The following sections describe some of these functions.

10.1.2.1 ASCII Function

The ASCII function returns the numeric ASCII value of a string's first character. The ASCII function returns an integer value from 0 to 255, inclusive. For instance, in the following example, the PRINT statement prints the integer value 66 because this is the ASCII value equivalent of an uppercase B, the first character in the string:

```
test_string$ = "BAT"
PRINT ASCII(test_string$)
END
```

Output

```
 66
```

Note that the ASCII value of a null string is zero.

10.1.2.2 CHR$ Function

The CHR$ function returns the character whose ASCII value you supply. If the ASCII integer expression that you supply is less than zero or greater than 255, HP BASIC treats it as a modulo 256 value. HP BASIC treats the integer expression as the remainder of the actual supplied integer divided by 256. Therefore, CHR$(325) is equivalent to CHR$(69) and CHR$(-1) is equivalent to CHR$(255).

The following program outputs the character whose ASCII value corresponds to the input value modulo 256:

```
PRINT "THIS PROGRAM FINDS THE CHARACTER WHOSE"
PRINT "VALUE (MODULO 256) YOU TYPE"
INPUT value%
PRINT CHR$(value%)
END
```

Output 1

```
THIS PROGRAM FINDS THE CHARACTER WHOSE
VALUE (MODULO 256) YOU TYPE
 ? 69
E
```

Output 2

```
THIS PROGRAM FINDS THE CHARACTER WHOSE
VALUE (MODULO 256) YOU TYPE
 ? 1093
E
```

10.1.3 String Numeric 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 a numeric string for floating-point constants.

The following sections describe some of the HP BASIC numeric string functions.

10.1.3.1 FORMAT$ Function

The FORMAT$ function converts a numeric value to a string. The output string is formatted according to a string you provide. The expression you give this function can be any string or numeric expression. The format string must contain at least one PRINT USING format field. The formatting rules are the same as those for printing numbers with PRINT USING. See Chapter 14 for more information about the PRINT USING statement and formatting rules.

```
A = 5
B$ = "##.##"
Z$ = FORMAT$(A, B$)
PRINT Z$
END
```

Output

```
 5.00
```

10.1.3.2 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 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:

```
PRINT NUM$(7465097802134)
PRINT NUM$(-50)
END
```

Output

```
 .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 following example shows the use of the NUM1$ function:

```
PRINT NUM1$(PI)
PRINT NUM1$(97.5 * 30456.23 + 30385.1)
PRINT NUM1$(1E-38)
END
```

Output

```
3.14159
2999870
.00000000000000000000000000000000000001
```

NUM1$ returns up to 6 digits of accuracy for SINGLE and SFLOAT real numbers, up to 16 digits of accuracy for DOUBLE numbers, up to 10 digits of accuracy for LONG integers, and up to 19 digits of accuracy for QUAD integers. NUM1$ returns up to 15 digits of accuracy for GFLOAT and TFLOAT numbers and up to 33 digits of accuracy for XFLOAT numbers.

The following example shows the difference between NUM$ and NUM1$:

```
A$ = NUM$(1000000)
B$ = NUM1$(1000000)
PRINT LEN(A$); "/"; A$; "/"
PRINT LEN(B$); "/"; B$; "/"
END
```

Output

```
 8 / .1E+07 /
 7 /1000000/
```

Note that *A$* has a leading and trailing space.

10.1.3.3 VAL% and VAL Functions

The VAL% function returns the integer value of a numeric string. This numeric string expression must be the string representation of an integer. It can contain the ASCII characters 0 to 9, a plus sign (+), or a minus sign (-).

The VAL function returns the floating-point value of a numeric string. The numeric string expression must be the string representation of some number. It can contain the ASCII characters 0 to 9, a plus sign (+), a minus sign (-), or an uppercase E.

The VAL function returns a number of the default floating-point data type. HP BASIC signals “Illegal number” (ERR=52) if the argument is outside the range of the default floating-point data type.

The following is an example of VAL and VAL%:

```
A = VAL("922")
B$ = "100"
C% = VAL%(B$)
PRINT A
PRINT C%
END
```

Output

```
 922
 100
```

10.1.4 String Arithmetic Functions

In BASIC, string arithmetic functions process numeric strings as arithmetic operands. This lets you add (SUM$), subtract (DIF$), multiply (PROD$), and divide (QUO$) numeric strings, and 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 more slowly than the corresponding 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 10–1 shows the string arithmetic functions and their formats, and gives brief descriptions of what they do.

Table 10–1 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%.

String arithmetic computations permit 56 significant digits. The functions QUO$, PLACE$, and PROD$, however, permit up to 60 significant digits. Table 10–2 shows how HP BASIC determines the precision permitted by each function and if that precision is implicit or explicit.

Table 10–2 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

10.1.4.1 SUM$ and DIF$ Functions

The SUM$ and DIF$ functions 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.

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.
- Trailing zeros are trunctated.

10.1.4.2 QUO$, PLACE$, and PROD$ Functions

In the QUO$, PLACE$, and PROD$ functions, the value of the integer expression argument explicitly determines numeric precision. That is, the integer expression parameter 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 point. For example, if the integer expression is 1, rounding occurs one digit to the right of the decimal point (the number is rounded to the nearest tenth). If the integer expression is 2, rounding occurs two digits to the right of the decimal point (the number is rounded to the nearest hundredth), and so on.
- For zero, rounding occurs to the nearest unit.
- For negative integer expressions, rounding occurs to the left of the decimal point. For example, if the integer expression is -1, rounding occurs one place to the left of the decimal point. In this case, HP 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;

HP BASIC moves the decimal point two places to the left, then rounds to units.

Note that when rounding numeric strings, HP BASIC returns only part of the number.

If the integer expression is between 5001 and 15,000, the following rules apply:

- If the integer expression is 10,000, HP BASIC truncates the number at the decimal point.
- If the integer expression is greater than 10,000 (10,000 plus *n*), HP BASIC truncates the numeric string *n* places to the right of the decimal point. For example, if the integer expression is 10,001 (10,000 plus 1), HP BASIC truncates the number starting one place to the right of the decimal point. If the integer expression is 10,002 (10,000 plus 2), HP BASIC truncates the number starting two places to the right of the decimal point, and so on.
- If the integer expression is less than 10,000 (10,000 minus *n*) HP BASIC truncates the numeric string *n* places to the left of the decimal point. For example, if the integer expression is 9999 (10,000 minus 1), HP BASIC truncates the number starting one place to the left of the decimal point. If the integer expression is 9998 (10,000 minus 2), HP BASIC truncates starting two places to the left of the decimal point, and so on.

The PLACE$ function returns a numeric string, truncated or rounded according to an integer argument you supply.

The following example displays the use of the PLACE$ function with several different integer expression arguments:

```
number$ = "123456.654321"
FOR I% = -5% TO 5%
    PRINT PLACE$(number$, I%)
NEXT I%
PRINT
FOR I% = 9995 TO 10005
    PRINT PLACE$(number$, I%)
NEXT I%
```

Output

```
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
```

The PROD$ function returns the product of two numeric strings. The returned string's precision depends on the value you specify for the integer precision expression.

```
A$ = "-4.333"
B$ = "7.23326"
s_product$ = PROD$(A$, B$, 10005%)
PRINT s_product$
END
```

Output

```
-31.34171
```

10.1.5 Date and Time Functions

HP BASIC supplies functions to return the date and time in numeric or string format. The following sections discuss these functions.

Note that you can also use certain system services and Run-Time Library routines for more sophisticated date and time functions. See the *HP OpenVMS System Services Reference Manual* and the *VMS Run-Time Library Routines Volume* for more information.

10.1.5.1 DATE$ Function

The DATE$ function returns a string containing a day, month, and year in the form *dd-Mmm-yy*. The date integer argument to the DATE$ function can have up to six digits in the form *yyyddd*, where *yyy* specifies the number of years since 1970 and *ddd* specifies the day of that year. If the numeric expression is zero, DATE$ returns the current date.

```
PRINT DATE$(0)
PRINT DATE$(126)
PRINT DATE$(6168)
END
```

Output

```
15-Jun-85
06-May-70
16-Jun-76
```

If you supply an invalid date (for example, day 370 of the year 1973), the results are undefined.

See Section 10.1.5.2 for the recommended replacement for DATE$, which has a two-digit year field in the result string.

10.1.5.2 DATE4$ Function

The DATE4$ function is strongly recommended as replacement for the DATE$ function to avoid problems in the year 2000 and beyond. It functions the same as the DATE$ function except that the year portion of the result string contains two more digits indicating the century. For example:

```
PRINT 32150, DATE$ (32150), DATE4$ (32150)
```

Produces the following output:

```
32150   30-May-02   30-May-2002
```

See the description of the DATE$ function for more information.

10.1.5.3 TIME$ Function

The TIME$ function returns a string displaying the time of day in the form *hh:mm* AM or *hh:mm* PM. 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:

```
PRINT TIME$(0)
PRINT TIME$(1)
PRINT TIME$(1440)
PRINT TIME$(721)
END
```

Output

```
03:53 PM
11:59 PM
12:00 AM
11:59 AM
```

10.1.5.4 TIME Function

The TIME function requests time and usage information from the operating system and returns it to your program. The information returned by the TIME function depends on the value of the argument passed to it. The values and the information they return are as follows:

Value	Information Returned
0	Returns the number of seconds elapsed since midnight
1	Returns the current job's CPU time in tenths of a second
2	Returns the current job's connect time in minutes
3	Returns zero
4	Returns zero

All other arguments to TIME are undefined and cause HP BASIC to signal "Not implemented" (ERR=250).

10.1.6 Terminal Control Functions

HP BASIC provides several terminal control functions. These functions let you:

- Enable and disable Ctrl/C trapping
- Enable and disable terminal echoing
- Read a single keystroke from a terminal

10.1.6.1 CTRLC and RCTRLC Functions

The CTRLC function enables Ctrl/C trapping, and the RCTRLC function disables Ctrl/C trapping. When Ctrl/C trapping is enabled, control is transferred to the program's error handler when Ctrl/C is detected at the controlling terminal.

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 previous contents.

For example, if you type Ctrl/C to the following 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.

```
WHEN ERROR USE error_handler
      Y% = CTRLC
   .
   .
   .
END WHEN
HANDLER error_handler
      IF ERR = 28 THEN PRINT #1%, "Abort"
   .
   .
   .
END HANDLER
```

Note

When you trap Ctrl/C with an error handler, your program might be in an inconsistent state; therefore, you should handle the Ctrl/C error and exit the program as quickly as possible.

10.1.6.2 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 screen.

If you specify channel #0 (your terminal) as the argument, 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.

If you do not use these functions, ECHO is the default. The following program shows a password routine in which the password does not echo:

```
Y% = NOECHO(0%)
INPUT "PASSWORD"; pword$
IF pword$=="PLUGH" THEN PRINT "THAT IS CORRECT"
END IF
Y% = ECHO(0%)
END
```

Note that the Y% = ECHO(0%) statement is necessary to turn the echo back on. If this statement were not included, then all subsequent user inputs would not echo to the terminal screen.

10.1.6.3 INKEY$ Function

The INKEY$ function reads a single keystroke from a terminal opened on a specified channel and returns the typed character.

If you specify a channel that is not open, HP BASIC signals the error "I/O channel not open" (ERR=9). If a file or a device other than a terminal is open on the channel, HP BASIC signals the error "Illegal operation" (ERR=141).

Once you have specified a channel, HP BASIC allows you to specify an optional WAIT clause. A WAIT clause followed by no value tells HP BASIC to wait indefinitely for input to become available. A WAIT clause followed by a value from 1 to 255 tells HP BASIC to wait the specified number of seconds for input.

```
DECLARE STRING
KEYSTROKE Inkey_Loop: WHILE 1%       KEYSTROKE = INKEY$(1%,WAIT)

      SELECT KEYSTROKE
          CASE '26'C
               PRINT "Ctrl/Z to exit"
               EXIT Inkey_Loop
          CASE CR,LF,VT,FF,ESC
               PRINT "Line terminator"
          CASE "PF1" TO "PF4"
               PRINT "P key"
          CASE "E1" TO "E6"
               PRINT "VT200 Function key"
          CASE "KP0" TO "KP9"
               PRINT "Application keypad key"
          CASE < SP
               PRINT "Control character"
          CASE '127'C
               PRINT "<DEL>"
          CASE ELSE
               PRINT "Character is "; KEYSTROKE
       END SELECT
NEXT
```

10.2 User-Defined Functions

The DEF statement lets you create your own single-line or multiline functions.

In HP BASIC, a function name consists of the following:

- The letters FN
- From 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 create user-defined functions using these function naming rules; however, it is recommended that you use explicit data typing when defining functions for new program development. See Chapter 12 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 multiline. Whether you use the single-line or multiline format for function definitions depends on the complexity of the function you create. In general, multiline DEF functions perform more complex functions than single-line DEF functions and are suitable for recursive operations.

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 255 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.

10.2.1 Single-Line DEF Functions

In a single-line DEF, the function name, the formal parameter list, and the defining expression all appear on the same line. The defining expression specifies the calculations that the function performs. You can pass up to 255 arguments to this function through the formal parameter list. These parameters are variables local to the function definition, and each formal parameter can be preceded by a data type keyword.

The following example creates a function named *fnratio*. This function has two formal parameters: *numer* and *denomin*, whose ratio is returned as a REAL value.

When the function is invoked, HP BASIC does the following:

- 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 (the PRINT statement)

The PRINT statement then prints the returned value.

```
DEF REAL fnratio (numer, denomin) = numer / denomin
PRINT fnratio(5.6, 7.8)
END
```

Output

```
 .717949
```

Note that the actual parameters you supply must agree in number and data type with those in the formal parameter list; you must supply numeric values for numeric variables, and string values for string variables.

The defining expression for a single-line function definition can contain any constant, variable, HP BASIC built-in function, or any user-defined function except the function being defined. The following examples are valid function definitions:

```
DEF FN_A(X) = X^2 + 3 * X + 4
DEF FN_B(X) = FN_A(X) / 2 + FN_A(X)
DEF FN_C(X) = SQR(X+4) + 1
DEF CUBE(X) = X ^ 3
```

Note that the name of the last function defined 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. The following function definition uses three HP 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 1 up to and including the character at position 2 (the day number).

- The VAL% function converts this resulting numeric string to an integer. In this way, *fnday_number* returns the day of the month as an integer.

```
DEF INTEGER fnday_number = VAL% (SEG$(DATE$(0%), 1%, 2%))
```

10.2.2 Multiline DEF Functions

The DEF statement can also define multiline functions. Multiline DEF functions are useful for expressing complicated functions. Note that multiline DEF functions 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 multiline DEF function contains DATA statements, they are global to the program unit.

Multiline function definitions can contain any constant, variable, HP BASIC built-in function, or user-defined function. In HP BASIC, the function definition can contain a reference to the function you are defining. Therefore, a multiline DEF function can be recursive, or invoke itself; however, HP BASIC does not detect infinitely recursive DEF functions during compilation. If your program invokes an infinitely recursive DEF function, HP BASIC will eventually signal a fatal run-time error, typically the error "Access violation."

You can use either the END DEF or EXIT DEF statements to exit from a user-defined function. The EXIT DEF statement is equivalent to an unconditional transfer to the END DEF statement.

The following example shows a multiline DEF function that uses both the EXIT and END DEF statements. The defining expression of the function is in the ELSE clause. This assigns a value to the function if *A* is less than 10. The second set of output shows what happens when *A* is greater than 10; HP 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. In this way, this example tests the arguments before the function is evaluated.

```
DEF fn_discount(A)
    IF A > 10
    THEN
       PRINT "OUT OF RANGE"
       EXIT DEF
    ELSE
       fn_discount = A^A
    END IF
END DEF

INPUT Z
PRINT fn_discount(Z)
END
```

Output 1

```
? 4
 256
```

Output 2

```
? 12
OUT OF RANGE
 0
```

If you do not explicitly declare the function with the DECLARE statement, the restrictions for naming a multiline DEF function are the same as those for the single-line DEF function; however, explicitly declaring a DEF function can make a program easier to read and understand. For instance, Example 1 concatenates two strings and Example 2 returns a number in a specified modulus.

```
DECLARE STRING FUNCTION concat (STRING, STRING) !Declare the function
   .
   .
   .
DEF STRING concat (STRING Y, STRING Z)
concat = Y + Z  !Define the function
FNEND
   .
   .
   .
new_string$ = concat(A$, B$) !Invoke the function
   .
   .
   .
END

DECLARE REAL FUNCTION mdlo (REAL, INTEGER)
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

  !If argument is negative, set flag negative% and set argument
  !to its absolute value.
  IF argument < 0
     THEN argument = ABS( argument )
        negative%  = -1%
  END IF
  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

  !Argument now contains the right number, but the sign might be wrong.
  !If the negative argument flag was set, make the result negative.

  IF negative%
     THEN mdlo = - argument
     ELSE mdlo = argument
  END IF

END DEF

INPUT "PLEASE INPUT THE VALUE AND THE MODULUS"; X,Y
PRINT mdlo(X,Y)
END
```

Output

```
PLEASE INPUT THE VALUE AND THE MODULUS? 7, 5
 2
```

Because these functions are declared in DECLARE statements, the function names do not have to conform to the traditional HP BASIC rules for naming functions.

Recursion occurs when a function calls itself. The following example defines a recursive function that returns a number's factorial value:

```
DECLARE INTEGER FUNCTION factor ( INTEGER )
DEF INTEGER factor ( INTEGER F )
   IF F <= 0%
      THEN factor = 1%
      ELSE factor = factor(F - 1%) * F
   END IF
END DEF
INPUT "INPUT N TO FIND N FACTORIAL"; N%
PRINT "N! IS"; factor(N%)
END
```

Output

```
INPUT N TO FIND N FACTORIAL? 5
N! IS 120
```

Any variable accessed or declared in the DEF function and not in the formal parameter list is global to the program unit. When HP 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 multiline DEF function except by invoking it. You should not transfer control out of a DEF function except by way of an EXIT DEF or END DEF statement. This means that:

- If the DEF function 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 function.
- An ON ERROR GO BACK statement can transfer control out of a DEF function; however, a RESUME statement in an error handler outside the DEF function cannot transfer control back into the DEF function.
- If the DEF function contains a handler, and was invoked from a protected region, an EXIT HANDLER statement causes control to be transferred to the specified handler for that protected region. However, if the DEF function contains a handler but was not invoked from a protected region, an EXIT HANDLER statement causes control to be transferred to the default error handler.

- A subroutine cannot be shared by more than one DEF function; however, if you rewrite the subroutine as a DEF function with no parameters, other function definitions can share it.

A DEF function never changes the value of a parameter passed to it. Also, because formal parameters are local to the function definition, you cannot access the values of these variables from outside the DEF statement. These variable names are known only inside the DEF statement.

In the following example, the variable *first* is declared only in the function *fn_sum*. When HP BASIC sees the second PRINT statement, it assumes that *first* is a new variable that is not declared in the main program. If you try to run this example, HP BASIC signals the error "Explicit declaration of first required." If you do not specify the OPTION TYPE = EXPLICIT statement, HP BASIC assumes that *first* is a new variable and initializes it to zero.

```
OPTION TYPE = EXPLICIT
DECLARE INTEGER A, B
DEF fn_sum(INTEGER first, INTEGER second) = first + second
A = 50
B = 25
PRINT fn_sum(A, B)
PRINT first
END
```

11
String Handling

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

11.1 Overview of Strings

A **string** is a sequence of ASCII characters. BASIC allows you to use the following types of strings:

- Dynamic strings
- Fixed-length strings
- String virtual arrays

Dynamic strings are strings whose length can change during program execution. The length of a dynamic string variable can change or not, depending on the statement used to modify it.

Fixed-length strings are strings whose length never changes. In other words, their length remains static. 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 statement. If a string variable is not part of a map or common block, RECORD, or virtual array, it is a dynamic string. When a string variable is fixed-length, its length does not change, regardless of the statement you use to modify it. Table 11–1 provides more information about string modification.

Strings in virtual arrays have both fixed-length and dynamic attributes. String virtual arrays have a specified maximum length from 0 to 512 characters. During program execution, the length of an element in a string virtual array can change; however, the length is always from 0 to the maximum string size specified when the array was created. See Section 11.4 and Chapter 13 for more information about virtual arrays.

Table 11–1 String Modification

Statement	Changes Made to Fixed-Length Strings	Changes Made to Dynamic Strings
LET	Value	Value and length
LSET	Value	Value
RSET	Value	Value
Terminal I/O Statements[1]	Value	Value and length

[1]Terminal I/O statements include INPUT, INPUT LINE, LINPUT, MAT INPUT, and so on.

11.2 Using Dynamic Strings

Although dynamic strings are less efficient than fixed-length strings, they are often more flexible. For example, to concatenate strings, you can use the LET statement to assign the concatenated value to a dynamic string variable, without having to be concerned about HP BASIC truncating the string or adding trailing spaces to it. However, if the destination variable is fixed-length, you must make sure that it is long enough to receive the concatenated string, or HP 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.

The LET, LSET, and RSET statements all operate on dynamic strings as well as fixed-length strings. The LET statement can change the length of a dynamic string; LSET and RSET do not. LSET and RSET are more efficient than LET when changing the value of a dynamic string. For more information about LSET and RSET, see Section 11.5.2 and Section 11.5.3.

In the following example, the first line assigns the value "ABC" to *A$*, the second line assigns "XYZ" to *B$*, and the third line assigns six spaces to *C$*. These variables are dynamic strings. In the fourth line, LSET assigns *A$* 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$* are assigned to *A$*. The fifth line uses LSET to assign *C$* the value of *A$* concatenated with *B$*. Because *C$* already has a length of 6, this statement assigns the value "ABCXYZ" to it.

```
LET A$ = "ABC"
LET B$ = "XYZ"
LET C$ = "      "
LSET A$ = A$ + B$
LSET C$ = A$ + B$
PRINT A$
PRINT C$
END
```

Output

```
ABC
ABCXYZ
```

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.

In this example, the first line assigns the null string to variable *A$*. The second line uses the LEN function to show that the null string has a length of zero. The third line uses the INPUT statement to assign a new value to *A$*, and the fourth and fifth lines print the new value and its length.

```
!Declare a dynamic string
LET A$ = ""
PRINT LEN(A$)
INPUT A$
PRINT A$
PRINT LEN(A$)
END
```

Output

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

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.

11.3 Using Fixed-Length Strings

It is generally more efficient to manipulate a fixed-length string than a dynamic string. Creating or modifying a dynamic string often causes HP BASIC to create new storage, and this increases processor overhead.

If a string variable is part of a map or common block, or virtual array, a LET, INPUT, LINPUT, or INPUT LINE statement changes its value, but not its length. In the following example, the MAP statement in the first line explicitly assigns a length to each string variable. Because the LINPUT statements cannot change this length, HP BASIC truncates values to fit the *address* and *city_state* variables. Because the *zip* variable is longer than the assigned value, HP BASIC left-justifies the assigned value and pads it with spaces. The sixth line uses the compile-time constant HT (horizontal tab) to separate fields in the employee record.

```
MAP (FIELDS) STRING full_name = 10,                              &
             address = 10,                                       &
             city_state = 10,                                    &
             zip = 10
LINPUT "NAME"; full_name
LINPUT "ADDRESS"; address
LINPUT "CITY AND STATE"; city_state
LINPUT "ZIP CODE"; zip
EMPLOYEE_RECORD$ = full_name + HT + address + HT &
                   + city_state + HT + zip
PRINT EMPLOYEE_RECORD$
END
```

Output

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

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

11.4 Using 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 13 for more information about virtual arrays.

Elements of string virtual arrays behave much like dynamic strings, with the following exceptions:

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

In the following example, the first two lines dimension a string virtual array and open a file on channel #1. The third line assigns a 10-character string to the first element of this string array, and to the variable *A$*. This 10-character string consists of "ABCDE" plus five null characters. The PRINT statements show that the length of *A$* is 10, while the length of *test(1)* is only 5 because HP BASIC strips trailing nulls from string array elements.

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

Output

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

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

11.5 Assigning String Data

To assign string data, you use the LET, LSET, RSET, and MID$ statements. The following sections describe how to use these statements.

11.5.1 LET Statement

The LET statement assigns string data to a string variable. The keyword LET is optional. Again, LSET is more efficient than LET when changing a dynamic string variable. In the following example, *B* is a string variable and "ret_status" is a quoted string expression:

```
LET B = "ret_status"
```

The LET statement changes the length of dynamic strings but does not change the length of fixed-length strings. The following example 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. The third line 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.

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

Output

```
ABC           10
XYZ           3
A             10
X             1
```

11.5.2 LSET Statement

The LSET statement left-justifies data and assigns it to a string variable, without changing the variable's length. In the following example, *ABC* is a string variable and "ABC" is a string constant:

```
LSET ABC = "ABC"
```

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. In the following example, the LET statement creates the 5-character string variable *test$*. The LSET statement in the second line assigns the string XYZ to the variable *test$* but does not change the length of *test$*. Because *test$* has a length of 5, the LSET statement pads the string XYZ with two spaces when assigning the value. The PRINT statement shows that test$ includes these two spaces.

```
LET test$ = "ABCDE"
LSET test$ = "XYZ"
PRINT "'"; test$; "'"
END
```

Output

```
'XYZ  '
```

LSET left-justifies a string expression longer than the string variable and truncates it on the right as shown in the following example:

```
LET test$ = "ABCDE"
LSET test$ = "12345678"
PRINT test$
END
```

Output

```
12345
```

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

11.5.3 RSET Statement

The RSET statement right-justifies data and assigns it to a string variable without changing the variable's length. In the following example, *C_R* is a string variable and "cust_rec" is a string constant:

```
RSET C_R = "cust_rec"
```

RSET right-justifies a string expression shorter than the string variable and pads it with spaces on the left. In the following example, the LET statement creates the 5-character string variable *test$*. The RSET statement in the second line assigns the string XYZ to *test$* 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 shows that *test$* includes these two spaces.

```
LET test$ = "ABCDE"
RSET test$ = "XYZ"
PRINT "'" ;  test$; "'"
END
```

Output

```
'  XYZ'
```

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 as shown in the following example:

```
LET test$ = "ABCDE"
RSET test$ = "987654321"
PRINT test$
END
```

Output

```
54321
```

The LET statement 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.

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

Output

```
'   ABC  '
```

11.5.4 MID$ Assignment Statement

You can replace a portion of a string with another string using the MID$ assignment statement. You specify a starting character position that indicates where to begin the substitution. If you specify a starting character position that is less than 1, HP BASIC assumes a starting character position of 1. In addition, you can optionally specify the number of characters to substitute from the source string expression. If you do not specify the number of characters to substitute, HP BASIC attempts to insert the entire source expression. However, the MID$ statement never changes the length of the target string variable; therefore, the entire source expression might not fit into the available space.

The following example shows the use of MID$ as an assignment statement. In this example, "ABCD" is the input string, the starting character position is 1, and the length of the segment to be replaced is 3 characters. Note that when you use MID$ as an assignment statement, the length of the input string does not change; therefore, the length of the result ("123D") is equal to the length of the input string.

```
DECLARE STRING old_string, replace_string
old_string = "ABCD"
replace_string = "123"
PRINT old_string
MID$(old_string,1,3) = replace_string
PRINT old_string
```

Output

```
ABCD
123D
```

Keep these considerations in mind when you use the MID$ assignment statement:

- The length argument is optional. If not specified, the number of characters replaced will be the minimum of the length of the replacement string and the length of the input string minus the starting position value.
- If the length of the segment is less than or equal to zero, HP BASIC assumes a length of zero.
- The length of the input string does not change regardless of the value of the length of the segment.

11.6 Manipulating String Data with String Functions

When used with the LET statement, HP 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)
- Extract segments from a string (SEG$, MID$)
- 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$)
- Edit a string (EDIT$)

These functions are discussed in the following sections. See the *HP BASIC for OpenVMS Reference Manual* for more information about each string's function.

11.6.1 LEN Function

The LEN function returns the number of characters in a string as an integer value. For example:

```
LEN(spec)
```

Spec is a string expression. The length of the string expression includes leading and trailing blanks. In the following example, the variable Z$ is set equal to "ABC XYZ", which has a length of eight:

```
alpha$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
PRINT LEN(alpha$)
Z$ = "ABC" + "  " + ""XYZ"
PRINT LEN(Z$)
END
```

Output

```
 26
 8
```

11.6.2 POS Function

The POS function searches a string for a group of characters (a substring). In the following example, *spec* is the string to be searched, *test* is the substring for which you are searching and *15* is the character position where HP BASIC starts the search:

```
POS(spec,test,15)
```

The position returned by POS 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 1, 2, 3, 4, or 5, HP BASIC still returns the value 5 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 as in the following example:

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

Output

```
 4
 0
```

If you specify a starting position other than 1, HP BASIC still returns the position of the substring relative to the beginning of the string as shown in the following example:

```
alpha$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Z$ = "HIJ"
PRINT POS(ALPHA$, Z$, 7%)
END
```

Output

```
 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 HP 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**. The following example 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.

```
DECLARE STRING CONSTANT table =    &
 "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
DECLARE STRING month, UPPER_CASE_MONTH, message
DECLARE INTEGER month_length
DECLARE REAL month_pos
PRINT "Please type the first three letters of a month"
PRINT "To end the program, type only Return";
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
END
```

Output

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

Keep these considerations in mind when you use POS:

- If you specify a starting position less than 1, POS assumes a starting position of one.
- If you specify a starting position greater than the searched string's length, POS returns a 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 returns the string's length plus one.
  - If the string to be searched is also null, POS returns a value of one.
  - If the specified starting position is less than or equal to 1, POS returns a value of one.
  - If the specified starting position is greater than one and less than or equal to the string's length plus 1, POS returns 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.

11.6.3 SEG$ Function

The SEG$ function extracts a segment (substring) from a string. The original string remains unchanged. In the following example, *time* is the input string, *13* is the position of the first character extracted, and *16* is the position of the last character extracted:

```
SEG$(time,13,16)
```

SEG$ extracts from the input string the substring that starts at the first character position, up to and including the last character position. It returns the extracted segment.

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

Output

```
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 the starting character position is less than 1, HP BASIC assumes a value of 1.
- If the starting character position is greater than the ending character position, or the length of the string, SEG$ returns a null string.
- If the ending character position is greater than the length of the string, SEG$ returns all characters from the starting character position to the end of the string.
- If the starting character position is equal to the ending character position, SEG$ returns the character at the starting position.

You can replace part of a string by using the SEG$ function with the string concatenation operator (+). In the following example, when HP 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.

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

Output

```
ABXYZFG

ABCDEFG
```

You can use similar string expressions to replace characters in any string. If you do not change the length of the target string, use the MID$ assignment statement to perform string replacement. A general formula to replace characters in positions *n* through *m* of string A$ with characters in B$ is as follows:

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

The following example replaces the sixth to ninth characters of the string "ABCDEFGHIJK" with "123456":

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

Output

```
ABCDE123456JK
ABCDEFGHIJK
123456
```

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

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

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

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

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

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

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

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

  C$ = SEG$(A$,1,LEN(A$)–*n*) + B$

- To insert B$ in A$ after the *n*th character in A$ use:

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

11.6.4 MID$ Function

The MID$ function extracts a specified substring, beginning at a specified character position and ending at a specified length. If you specify a starting character position that is less than 1, HP BASIC automatically assumes a starting character position of 1.

In the following example, the MID$ function uses the input string “ABCD”, and extracts a segment consisting of 3 characters. Because HP BASIC automatically assumes a starting character position of 1 when the specified starting character position is less than 1, the string that is extracted begins with the first character of the input string.

```
DECLARE STRING old_string, new_string
old_string = "ABCD"
new_string = MID$(old_string, 0, 3)
PRINT new_string
```

Output

```
ABC
```

Keep these considerations in mind when you use MID$:

- If the position of the segment's first character is greater than the input string, MID$ returns a null string.
- If the length of the segment is greater than the length of the input string, HP BASIC returns the string that begins at the specified starting character position and includes all characters remaining in the string.
- If the length of the segment is less than or equal to zero, MID$ returns a null string.
- If you specify a floating-point expression for the position of the segment's first character or for the length of the segment, HP BASIC truncates it to a long integer.

11.6.5 STRING$ Function

The STRING$ function creates a character string containing multiple occurrences of a single character. In the following example, 23 is the length of the returned string, and 30 is the ASCII value of the character that makes up the string. This value is treated modulo 256.

```
STRING$(23,30)
```

The following example creates a 10-character string containing uppercase As, which have ASCII value 65:

```
out$ = STRING$(10%, 65%)
PRINT out$
END
```

Output

```
AAAAAAAAAA
```

Keep these considerations in mind when you use STRING$:

- If the length of the returned string is less than or equal to zero, STRING$ returns a null string.
- If the length of the returned string is greater than 65535, HP BASIC signals an error.

11.6.6 SPACE$ Function

The SPACE$ function creates a character string containing spaces. In this example, *5* is the number of spaces in the string:

```
SPACE$(5)
```

The following example creates a 9-character string which contains 3 spaces:

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

Output

```
ABC   XYZ
```

11.6.7 TRM$ Function

The TRM$ function removes trailing blanks and tabs from a string. The input string remains unchanged. In the following example, all trailing blanks that appear in the string expression "ABCDE " are removed once the TRM$ function is invoked:

```
A$ = "ABCDE   "
B$ = "XYZ"
first$ = A$ + B$
second$ = TRM$(A$) + B$
PRINT first$
PRINT second$
END
```

Output

```
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).

11.6.8 EDIT$ Function

The EDIT$ function performs one or more string editing functions, depending on the value of an argument you supply. The input string remains unchanged. In the following example, *stu_rec* is a string expression and *32* determines the editing function performed:

```
EDIT$(stu_rec,32)
```

Table 11–2 shows the action HP BASIC takes for a given value of the integer expression.

Table 11–2 EDIT$ Options

Value of Expression	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, HP BASIC suppresses all editing for the characters following the quotation mark.

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

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

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

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

Output

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

11.7 Manipulating String Data with Multiple Maps

Mapping a string storage area in more than one way lets you extract a substring from a string or concatenate strings. In the following example, the three MAP statements reference the same 108 bytes of data:

```
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
```

You can move data into a map in different ways. For instance, you can use terminal input, arrays, and files. In the following example, the READ and DATA statements are used to move data into a map:

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

Because all the MAP statements in the previous example reference the same storage area *(emprec)*, you can access parts of this area through the mapped variables as shown in the following examples:

Example 1

```
PRINT full_name$
PRINT wage_class$
PRINT salary_ytd$
```

Output 1

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

Example 2

```
PRINT last_name$
PRINT tax_ytd$
```

Output 2

```
DAVIDSON
$925.31
```

You can assign a new value to any of the mapped variables. The following example prompts the user for changed information by displaying a menu of topics. The user can then choose which topics need to be changed and then separately assign new values to each variable.

```
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
END
```

Output

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

See Chapter 7 and the *HP BASIC for OpenVMS Reference Manual* for more information about the MAP statement.

12
Program Segmentation

This chapter describes how to:

- Declare HP BASIC subprograms
- Write HP BASIC subprograms

Program segmentation is the process of dividing a program into small, manageable routines and modules. In a segmented or modular program, each routine or module usually performs only one logical function. You can, therefore, design and implement a modular program faster than a nonsegmented program. Program modularity also simplifies debugging and testing, as well as program maintenance and transportability.

Subprograms processed by the HP BASIC compiler conform to the OpenVMS Procedure Calling Standard. This standard prescribes how arguments are passed, how values are returned, and how procedures receive and return control. Because HP BASIC conforms to the OpenVMS Procedure Calling Standard, an HP BASIC subprogram or main program can call or be called by any procedure written in a language that also conforms to this standard. For information about calling non-BASIC procedures, see Chapter 19.

12.1 HP BASIC Subprograms

HP BASIC has SUB and FUNCTION subprograms. Each of these subprograms receives parameters and can modify parameters passed by reference or by descriptor. The differences between SUB and FUNCTION subprograms are as follows:

- FUNCTION subprograms must be declared with an EXTERNAL statement in the calling program. Declaring SUB subprograms is optional.
- FUNCTION subprograms return a value; SUB subprograms do not return a value.

All subprograms invoked by an HP BASIC program must have unique names. A HP BASIC program cannot have different subprograms with the same identifiers.

Subprograms can return a value to the calling program with parameters. You can use subprograms to separate routines that you commonly use. For example, you can use subprograms to perform file I/O operations, to sort data, or for table lookups.

You can also use subprograms to separate large programs into smaller, more manageable routines, or you can separate modules that are modified often. If all references to system-specific features are isolated, it is easier to transport the program to a different system. OpenVMS System Services and OpenVMS Run-Time Library routines are specific to OpenVMS systems; therefore, you should consider isolating references to them in subprograms. Chapter 19 describes how to access Run-Time Library routines and system services from HP BASIC.

You should also consider isolating complex processing algorithms that are used commonly. If complex processing routines are isolated, they can be shared by many programs while the complexity remains hidden from the main program logic. However, they can share data only if the following is true:

- Data is passed as a parameter from the CALL statement or function invocation to the subprogram—see Section 12.2 for more information.
- Data is part of a MAP or COMMON block—see Chapter 6 for information about using MAP and COMMON statements.
- Data is in a file—see Chapter 13 for more information about accessing data from a file.

All DATA statements are local to a subprogram. Each time you call a subprogram, HP BASIC positions the data pointer at the beginning of the subprogram's data.

The data pointer in the main program is not affected by READ or RESTORE statements in the subprogram (in contrast with the RESTORE # statement, which resets record pointers to the first record in the file no matter where it is executed). Chapter 5 contains more information about the READ and RESTORE statements. For more information about the RESTORE # statement, see Chapter 13.

12.1.1 SUB Subprograms

A SUB subprogram is a program module that can be separately compiled and that cannot return a value. A SUB subprogram is delimited by the SUB and END SUB statements. You may use the EXTERNAL statement to explicitly declare the SUB subprogram.

The END SUB statement does the following:

- Marks the end of the SUB subprogram
- Does not affect I/O operations or files
- Releases the storage allocated to local variables
- Returns control to the calling program

The EXIT SUB statement transfers control to the statement lexically following the statement that invoked the subprogram. It is equivalent to an unconditional branch to an END SUB statement.

The following SUB subprogram sorts two integers. If this SUB statement is invoked with actual parameter values that are already in sorted order, the EXIT SUB statement is executed and control returns to the calling program.

```
SUB sort_out (INTEGER X, INTEGER Y)
DECLARE INTEGER temp
  IF X > Y
   THEN
      temp = X
      X = Y
      Y = temp
   ELSE
      EXIT SUB
   END IF
END SUB
```

12.1.2 FUNCTION Subprograms

A FUNCTION subprogram is a program module that returns a value and can be separately compiled. It must be delimited by the FUNCTION and END FUNCTION statements. You use the EXTERNAL statement to name and explicitly declare the data type of an external function.

The END FUNCTION statement does the following:

- Marks the end of a function subprogram
- Does not affect I/O operations or files
- Releases the storage allocated to local variables
- Optionally specifies a return value for the function
- Returns control to the calling program

The EXIT FUNCTION statement immediately returns program control to the statement that invoked the function and optionally returns the function's return value. It is equivalent to an unconditional transfer to the END FUNCTION statement.

You can specify an expression with both the END FUNCTION and EXIT FUNCTION statements, which is another way of returning a function value. This expression must match the function data type, and it supersedes any function assignment. For more information, see the *HP BASIC for OpenVMS Reference Manual*.

The following 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.

```
FUNCTION REAL Sphere_volume (REAL R)
     IF R <= 0
       THEN
           Sphere_volume = 0.0
       ELSE
           Sphere_volume = 4/3 * PI * R ** 3
      END IF
END FUNCTION
```

The following example declares the FUNCTION subprogram and invokes it:

```
PROGRAM call_sphere
  EXTERNAL REAL FUNCTION SPHERE_VOLUME(REAL)
  PRINT  SPHERE_VOLUME(5.925)
END PROGRAM
```

Note that this module is compiled separately from the FUNCTION subprogram. You can link these modules together to run the program from DCL level.

12.2 Declaring Subprograms and Parameters

You declare a subprogram by naming it in an EXTERNAL statement in the calling program. You may also declare the data type of each parameter. If the subprogram is a function, the EXTERNAL statement also lets you specify the data type of the returned value.

The following statements are subprogram declarations using the EXTERNAL statement:

```
EXTERNAL SUB my_sub (LONG, STRING)
EXTERNAL GFLOAT FUNCTION my_func (GFLOAT, LONG, GFLOAT)
EXTERNAL REAL FUNCTION determinant (LONG DIM(,))
```

Note that the parameter lists contain only data type and dimension information; they cannot contain any format or actual parameters. When the external procedure is invoked, HP BASIC ensures that the actual parameter data type matches the data type specified in the EXTERNAL declaration. However, HP BASIC does not check to make sure that the parameters declared in the EXTERNAL statement match those in the external routine. You must ensure that these parameters match.

You can pass data of any HP BASIC data type to an HP BASIC subprogram, including RFAs and RECORDs. HP BASIC allows you to pass up to 255 parameters, separated by commas. The data can be any one of the following:

- Constants
- Variables
- Expressions
- Functions
- Array elements
- Entire arrays (but not virtual arrays)

For passing constants, variables, functions, and array elements, name them in the argument list. For example:

```
CALL SUB01(var1, var2)

CALL SUB02(Po_num%, Vouch, 66.67, Cust_list(5), FNA(B%))
```

However, when passing an entire array, you must use a special format. You specify the array name followed by commas enclosed in parentheses. The number of commas must be the number of array dimensions minus one. For example, *array_name()* is a one-dimensional array, *array_name(,)* is a two-dimensional array, *array_name(,,)* is a three-dimensional array, and so on.

The following example creates a three-dimensional array, loads the array with values, and passes the array to a subprogram as a parameter. The subprogram can access and change values in array elements, and these changes remain in effect when control returns to the main program.

```
PROGRAM fill_array
OPTION TYPE = EXPLICIT
DECLARE LONG I,J,K, three_d(10,10,10)
EXTERNAL SUB example_sub (LONG DIM(,,))
FOR I = 0 TO 10
    FOR J = 0 TO 10
        FOR K = 0 TO 10
            three_d(I,J,K) = I + J + K
        NEXT K
    NEXT J
NEXT I

CALL  example_sub( three_d(,,))
END PROGRAM

SUB example_sub( LONG X( , , ))
   .
   .
   .
END SUB
```

If you do not specify data types for parameters, the default data type is determined by:

- The last specified parameter data type
- An OPTION statement
- An HP BASIC compilation qualifier (for example, /REAL_SIZE=DOUBLE)
- The system default

The last specified parameter data type overrides all the other default data types, the defaults specified in the OPTION statement override any compilation qualifiers and system defaults, and so on. When you know the length of a string or the dimensions of an array at compile time, you can achieve optimum performance by passing them BY REF. When you call programs written in other languages, the practice of declaring subprograms and specifying the data types of parameters becomes more important because other languages might not use the HP BASIC default parameter-passing mechanisms. For more information about calling subprograms written in other languages, see Chapter 19.

12.3 Compiling Subprograms

an HP BASIC source file can contain multiple program units. When you compile such a file, HP BASIC produces a single object file containing the code from all the program units. You can then link this object file to create an executable image.

If the main program and subprograms are in separate source files, you can compile them separately from the DCL level. The following command causes HP BASIC to create MAIN.OBJ, SUB1.OBJ, and SUB2.OBJ by separating the file names with commas:

```
$ BASIC main,sub1,sub2
```

To link these programs, you must specify all object files as input to the OpenVMS Linker.

Alternatively, you can compile multiple modules into a single object file at the DCL command level by separating the file names with a plus sign (+) as follows:

```
$ BASIC main+sub1+sub2
```

The plus signs used to separate the file names instruct HP BASIC to create a single object file called MAIN.OBJ from the three source modules. To link this program, you specify only one input file to the linker.

When creating a multiple-unit program, follow these rules:

- If the source file contains line numbers, then the line numbers for each subprogram must be numerically greater than the highest line number of all preceding subprograms.
- Line numbers must be unique and no greater than 32767.
- Each subprogram must end with an END SUB or END FUNCTION statement before the next subprogram begins.
- If the source file contains line numbers, then text following an END SUB or END FUNCTION statement must begin on a numbered line.
- If the source file does not contain line numbers, then text following an END SUB or END FUNCTION statement must begin on a new physical line.

Note that in a multiple-unit program that contains line numbers, any comments or statements following an END, END SUB, or END FUNCTION statement become part of the preceding subprogram unless they begin on a numbered line. In a multiple-unit program that does not contain line numbers,

however, any comments following an END, END SUB, or END FUNCTION statement become part of the following subprogram if one exists.

In the following example, the function *Strip* changes all brackets to parentheses in the string *A$* or *alpha*, and strips all trailing spaces and tabs:

```
PROGRAM scan
  EXTERNAL STRING FUNCTION Strip (STRING)
  A$ = "USER$DISK:[BASIC.TRYOUTS]"
  B$ = Strip( A$ )
  PRINT B$
END PROGRAM

FUNCTION STRING Strip( STRING alpha )
IF (POS( alpha, "[", 1%)) > 0%
   THEN Strip = EDIT$(alpha, 128% +64%)
   ELSE Strip = EDIT$(alpha, 128%)
END IF
END FUNCTION
```

12.4 Invoking Subprograms

The following sections describe how to invoke subprograms and pass parameters to subprograms.

12.4.1 Invoking SUB Subprograms

The CALL statement transfers control to a subprogram, and optionally passes arguments to it. The parameters in the CALL statement specify variables, constants, expressions, array elements, or entire arrays to be passed to the subprogram. You can also specify a function in the argument list. HP BASIC passes the value returned by the function to the subprogram. If possible, HP BASIC converts the actual arguments to the data type specified in the EXTERNAL statement. HP BASIC signals an error when the conversion is not possible.

The following example shows an HP BASIC main program calling a BASIC subprogram. The main program prompts for three integers: *A*, *B*, and *C*. It then passes these variables as parameters to the SUB subprogram. The subprogram prints the sum of these variables and returns control to the calling program.

```
PROGRAM get_input
  OPTION TYPE = EXPLICIT
  EXTERNAL SUB SUB01(LONG, LONG, LONG)
  DECLARE LONG A, B, C
  INPUT "Please type three integers"; A, B, C
  CALL SUB01 (A, B, C)
END PROGRAM

SUB SUB01 (LONG X, LONG Y, LONG Z)
  PRINT "The sum is"; X + Y + Z
END SUB
```

12.4.2 Invoking FUNCTION Subprograms

The following example performs the same task as the SUB program; however, this example uses a FUNCTION subprogram that returns the value to the main program and the main program prints the result:

```
PROGRAM invoke_funct
  EXTERNAL LONG FUNCTION FUN01(LONG, LONG, LONG)
  DECLARE LONG A, B, C
  INPUT "Please type three integers"; A, B, C
  PRINT "The sum is"; FUN01(A, B, C)
END PROGRAM

FUNCTION LONG FUN01 (LONG X, LONG Y, LONG Z)
  FUN01 =  X + Y + Z
END FUNCTION
```

If you do not assign a value to the function name and you do not specify a return value on an EXIT FUNCTION or END FUNCTION statement, the function returns zero or the null string.

Note that when writing FUNCTION subprograms, you must specify a data type for the function in both the main program EXTERNAL statement and the subprogram FUNCTION statement. This data type keyword specifies the data type of the value returned by the function subprogram. You should ensure that the data type specified in an EXTERNAL FUNCTION statement matches the data type specified in the FUNCTION statement.

If you declare a FUNCTION subprogram with an EXTERNAL statement and use the CALL statement to invoke the function, it executes correctly but the function value is not available. Note that BASIC still performs parameter validation when you invoke a function with the CALL statement.

Note that you cannot use the CALL statement to invoke a string or packed decimal function.

12.5 Returning Program Status

A PROGRAM unit lets you return a status from an HP BASIC image by optionally including an integer expression with the END PROGRAM and EXIT PROGRAM statements. After executing a program, you can examine this status by checking the DCL symbol $STATUS. By default, HP BASIC returns a status of 1, indicating success. Success is signaled with an odd numbered status value, while an error is signaled with an even numbered value. $STATUS contains the same value as the integer expression for the exit status in the EXIT and END PROGRAM statements. Note that if a program is terminated with an EXIT PROGRAM statement, the expression on the EXIT PROGRAM statement overrides any expression on the END PROGRAM statement.

In the following example, *exit_status* contains the status value returned by the program. After program execution, $STATUS has the value of *exit_status*. You can examine the value of $STATUS and display the corresponding message text with the lexical function F$MESSAGE at DCL level, as shown in the following example:

```
PROGRAM Venture
  DECLARE INTEGER exit_status,                    &
          REAL capital
  EXTERNAL LONG CONSTANT SS$_BADPARAM
  EXTERNAL SUB play_safe(INTEGER),                &
           minor_risk(INTEGER),major_risk(INTEGER)
  Exit_status = 1%
  SET NO PROMPT
  How_much:
  INPUT "Enter the amount of your free capital $";capital
  SELECT capital
     CASE = 0
             exit_status = SS$_BADPARAM
             EXIT PROGRAM exit_status
     CASE < 5000
             CALL play_safe(capital)
     CASE < 15000
             CALL minor_risk(capital)
     CASE < 50000
             CALL major_risk(capital)
     CASE ELSE
             PRINT "I can't cope with that amount, try again."
  END SELECT
  GOTO How_much
    .
    .
    .
END PROGRAM exit_status
```

After program execution, you can examine the status of the program at DCL level:

```
$ SHOW SYMBOL $STATUS
$ STATUS = "%X10"
$ error_text = F$MESSAGE(%X10)
$ SHOW SYMBOL error_text
ERROR_TEXT = "SYSTEM-W-BADPARAM, bad parameter value"
```

The PROGRAM statement is always optional; EXIT PROGRAM and END PROGRAM are legal without a matching PROGRAM statement. Without a PROGRAM statement, these statements still exit the main compilation unit. The EXIT PROGRAM and END PROGRAM statements are not valid within SUB, FUNCTION, or PICTURE subprograms.

13
File Input and Output

This chapter explains BASIC file organizations and record operations that are implemented through OpenVMS Record Management Services (RMS). For a more thorough understanding of file organization and file and record operations, see the *OpenVMS Record Management Services Reference Manual*.

RMS stores data in physical **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 18 to 8192 bytes.

RMS stores one or more **data records** in each block. A data record can also be divided into smaller units, called **fields**. A data record can be smaller than, equal to, or larger than a disk block.

13.1 Record Formats

The format of a record determines how RMS stores the record in a block. You specify the record format in an OPEN statement. The following are valid BASIC record formats:

- Fixed-length records
- Variable-length records
- Stream records

13.1.1 Fixed-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; this process 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 or stream records.

13.1.2 Variable-Length Records

Variable-length records can have different lengths, but no record can exceed a maximum size set for the file. When the record is written to a file, RMS adds a record length header that contains 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 than fixed-length records, manipulation of the record length headers generates processor overhead.

13.1.3 Stream Records

BASIC interprets stream records as a continuous sequence, or stream, of bytes. Unlike the fixed- and variable-length formats, stream records do not contain control information such as record counts, segment flags, or other system-supplied boundaries. Stream records are delimited by special characters or character sequences called **terminators**. Note that stream record formats are valid only in sequential files.

RMS defines the following types of stream record formats:

- STREAM records can be delimited by any special character (usually a carriage return/line-feed pair).
- STREAM_LF records must be delimited by a line-feed character.
- STREAM_CR records must be delimited by a carriage return.

While you can access existing files of any one of these stream record formats, BASIC creates new stream files only in the STREAM format; you can create files of the other two stream record formats by modifying the RMS FAB control structure in a USEROPEN routine. For more information about USEROPEN routines, see Section 13.8.11.

13.2 File Organizations

HP BASIC provides the following file organizations:

- Terminal-format
- Sequential
- Relative
- Indexed
- Virtual

If you do not specify a file organization when creating a file, the default is a terminal-format file (a sequential file with variable-length records). The following sections describe each type of file organization.

13.2.1 Terminal-Format Files

A **terminal-format file** is a sequential file of variable-length records. Terminal-format files are the default; that is, you create a terminal-format file when you do not specify a file organization when you open a file. You can then use the PRINT, INPUT, INPUT LINE, and LINPUT statements to access a terminal-format file. See Chapter 5 and Chapter 6 for more information about terminal-format files.

13.2.2 Sequential Files

A **sequential file** contains records that are stored in the order they are written. Sequential files can contain records of any valid BASIC record format: fixed-length, variable-length, or stream. You usually read a sequential file from the beginning; therefore, a sequential file is most useful when you access the data sequentially each time you use it. You can also access sequential fixed-length records randomly by specifying a record number if the file resides on disk. In either case, sequential files can reside on both disk and magnetic tape devices, and those stored on disk support shared access.

13.2.3 Relative Files

A **relative file** contains a series of cells that are numbered consecutively from 1 to *n*, where *n* represents the relative record number. Each cell can contain only a single record. For fixed-length records, the length of each cell equals the record length plus 1 byte. For variable-length records, the length of the cell equals the maximum record size plus 3 bytes.

You can access records in a relative file either sequentially or randomly. The relative record number is the key value in random access mode; that is, to access a record in a relative file in random access mode, you must know the relative record number of that record. You can add records to a relative file either at the end of the file or into any empty cell.

Relative files are most useful when randomly accessed and when the record can be identified by its cell number (for example, when inventory numbers correspond to cell numbers). Relative files support shared access. You can delete records from relative files, but not sequential files.

13.2.4 Indexed Files

An **indexed file** contains data records that are sorted in ascending or descending order according to a primary index key value. The **index key** is a record field (or set of fields) that determines the order in which the records are logically accessed. Keys must be variables declared in a MAP statement. Keys can be any one of the following:

- Strings
- WORD integers
- LONG integers
- Quadword integers
- Packed decimal numbers

String keys can also be segmented; the key can be composed of up to eight string variables in a map. Quadword keys must be referenced using a record or group exactly 8 bytes long.

Along with the primary index key value, you can also specify up to 254 alternate keys; RMS creates one index for each key you specify. For each of these keys you can also specify either an ascending or descending collating sequence. Each index is stored as part of the file, and each entry in the index contains a pointer to a record. Therefore, each key you specify corresponds to a sorted list of record pointers.

An indexed file of library books, for example, might be ordered by book title; that is, the title of the book is the primary key for the file. The keys for alternate indexes might include the author's name and the book's Library of Congress number. Neither of these alternate indexes contains the actual records; instead, they contain sorted pointers to the appropriate records.

Indexed files are most useful when randomly accessed or when you want to access the records in more than one way.

13.2.5 Virtual Files

A **virtual file** is a random access file that stores one or more data records or virtual array elements in each physical 512-byte disk block. You create a virtual file by specifying ORGANIZATION VIRTUAL as part of the OPEN statement. Apart from virtual arrays and compatibility with BASIC and BASIC-PLUS-2, you should use sequential fixed-length instead of virtual files, as they provide the same capabilities. See Section 13.5 for more information about accessing the individual records in a disk block.

13.3 Record Access and Record Context

Record access modes determine the order in which your program retrieves or stores records in a file. They determine the record context: the current record and the next record to be processed. When your program successfully executes any record operation, the current record and next record pointers can change. If a record operation is unsuccessful, these pointers do not change.

The record access modes valid for RMS are:

- Sequential access—valid on any file organization
- Random-by-record number access—valid on sequential fixed and all relative files
- Random-by-key access—valid on indexed files
- Random-by-RFA (Record File Address) access—valid on any RMS file located on disk

With sequential access, the next record is the next logical record in the file. In the case of relative files, the next logical record is the next existing record (deleted or never-written records are skipped). In the case of indexed files, the next logical record is the record with the next ascending or descending value in the current key of reference depending on that key's collating sequence. You can therefore access relative or indexed files sequentially by not specifying a relative record number or key value.

You can also access sequential fixed-length and relative files randomly by record number; that is, you can specify the record number of the record to be retrieved. For relative files, this record number corresponds to the cell number of the desired record.

You can access indexed files randomly by key. The key specification includes a primary or alternate key and its value. BASIC retrieves the record corresponding to that value in the particular key chosen.

You can access disk files of any organization by Record File Address (RFA); this means that you specify an RFA variable whose value uniquely identifies a particular record. The RFA requires six bytes of information. For more information about RFAs, see Section 13.6.10.

13.4 I/O and Record Buffers

An **I/O buffer** is a storage area in your program that RMS uses to store data for I/O operations. You do not have direct access to I/O buffers; they are controlled entirely by RMS. The I/O buffer holds blocks of data transferred from the device, and its size is always greater than or equal to that of the record buffer. For more information about the amount of storage allocated for I/O buffers, see the *OpenVMS Record Management Services Reference Manual*.

A **record buffer** is another storage area in your program. You have direct access to and control of the record buffer. When your program reads a record from a file, the information is transferred from the file to the I/O buffer in one large chunk of data, and then the requested record is transferred 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 either when the I/O buffer is full or when other blocks need to be read in.

You can use MAP statements to create **static record buffers** and associate program variables with areas (fields) of the buffer. Static record buffers are buffers whose size does not change during program execution and whose program variables are always associated with the same fields in the buffer.

You can create **dynamic record buffers** with either a MAP DYNAMIC or a REMAP statement. These statements, when used after a MAP statement, associate or reassociate a particular program variable with a different area (field) of the record buffer; however, the total size of a record buffer does not change during program execution.

Note

If you do not 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; however, MOVE statements do not transfer data to or from a file.

13.5 Accessing the Contents of a Record

HP BASIC provides the following methods for accessing the contents of a record:

- MAP statement
- MAP DYNAMIC and REMAP statements (dynamic mapping)

- MOVE statements
- FIELD statements

The FIELD statement is a declining feature and is not recommended for new program development. It is recommended that you use either MAP statements, dynamic mapping, or MOVE statements to access record contents.

13.5.1 MAP Statement

Normally, a record is divided into predetermined fields, the sizes of which are known at compile time. The MAP statement creates the storage area for this record and determines its total size. The following examples show how the MAP statement creates the record storage area:

Example 1

```
RECORD name_addr
  STRING last_name = 15,    &
         street_name = 30,  &
  INTEGER house_num
END RECORD
MAP (student_buffer) name_addr student_info
```

Example 2

```
MAP (Emp_rec)
    STRING Emp_name = 25,                   &
    LONG Badge,                             &
    STRING Address = 25,                    &
    STRING Department = 4
```

13.5.2 MAP DYNAMIC and REMAP Statements

There are situations where predetermined fields are not applicable or possible. In these situations, you must perform record defielding in your program at run time. Using the MAP DYNAMIC statement, you can specify the variables in the map whose positions can change at run time. The REMAP statement then specifies the new positions of the variables named in the MAP DYNAMIC statement.

The following example shows how you can use MAP, MAP DYNAMIC, and REMAP to deblock your record fields. The MAP statement allocates a storage area of 2048 bytes and names it *Emp_rec*. The MAP DYNAMIC statement specifies that the variables *Emp_name*, *Badge*, *Address*, and *Department* are all located in *Emp_rec*, and that their positions can be changed at run time with the REMAP statement. The REMAP statement then redefines these variables to their appropriate sizes.

```
MAP (Emp_rec) FILL$ = 2048

MAP DYNAMIC (Emp_rec)                                   &
    STRING Emp_name,                                    &
    LONG Badge,                                         &
    STRING Address,                                     &
    STRING Department

REMAP (Emp_rec) FILL$ = Record_offset,                  &
    Emp_name = 25,                                      &
    Badge,                                              &
    Address = 25,                                       &
    Department = 4
```

Note that when accessing virtual or sequential files, you can specify a RECORD clause for the GET statement. The following example opens a virtual file with each block containing 512 bytes. However, each block contains 4 logical records that are 128 bytes long. Each of these logical records consists of a 20-character first name field, a 44-character last name field, and a 64-character company name field.

```
DECLARE WORD Record_number
MAP (Virt) STRING FILL = 512
MAP DYNAMIC (Virt) STRING First_name,        &
                   Last_name,               &
                   Company
OPEN "VIRT.DAT" FOR INPUT AS FILE #5,        &
             VIRTUAL, MAP Virt
Record_number = 1%
WHEN ERROR IN
  WHILE -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
USE
   IF ERR = 11%
         THEN
            PRINT "Finished"
            CONTINUE 32767
         ELSE EXIT HANDLER
         END IF
END WHEN
END
```

After the first 512-byte block is brought into memory, the FOR...NEXT loop deblocks the data 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 block.

For more information about the MAP DYNAMIC and REMAP statements, see Chapter 7 and the *HP BASIC for OpenVMS Reference Manual*.

13.5.3 MOVE Statement

The MOVE statement defines data fields and moves them to and from the record buffer created by HP BASIC. For example:

```
MOVE FROM #9%, A$, Cost, Name$ = 30%, ID_num%
```

This statement moves a record with four data fields from the record buffer to the variables in the list as follows:

- A string field *A$* with a default length of 16 characters
- A number field *Cost* of the default data type
- A second 30-character string field *Name$*
- An integer field *ID_num%*

Valid variables in the MOVE list are:

- Scalar variables
- Arrays
- Array elements
- FILL items

Because BASIC dynamically assigns space for string variables, the default string length during a MOVE TO operation 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 $n - 1$ commas, where n is the number of dimensions in the array. Note that these commas must be enclosed in parentheses. You specify a single array element by naming the array and the subscripts of that element. The following statement moves three arrays from the program to the record buffer. *A$* specifies a 1-dimensional string array, *C* specifies a 2-dimensional array of the default data type, and *D%* specifies a 3-dimensional integer array. *B*(3,2) specifies the element of array *B* that appears in row 3, column 2.

```
MOVE TO #5%, A$(), C(,), D%(,,), B(3,2)
```

Successive MOVE statements to or from the buffer start at the beginning of the record buffer. If a MOVE TO operation only partially fills the buffer, the rest of the buffer is unchanged. You use the GET statement to read a record from a file, and then you move the data from the buffer to variables and reference the variables in your program. A MOVE TO operation moves data from the variables into the buffer created by HP BASIC. A PUT or UPDATE statement then moves the data from the buffer to the file.

The following program opens file MOV.DAT, reads the first record into the buffer, and moves the data from the buffer into the variables specified in the MOVE FROM statement:

```
DECLARE STRING Emp_name, Address, Department
DECLARE LONG Badge

OPEN "MOV.DAT" AS FILE #1%,            &
     RELATIVE VARIABLE,                &
     ACCESS MODIFY, ALLOW NONE,        &
     RECORDSIZE 512%
GET #1%
MOVE FROM #1%,                         &
     Emp_name = 25,                    &
     Badge,                            &
     Address = 25,                     &
     Department = 4

   .
   .
   .
MOVE TO #1%,                            &
     Emp_name = 25,                     &
     Badge,                             &
     Address = 25,                      &
     Department = 4

UPDATE #1%
CLOSE #1%
END
```

The MOVE TO statement moves the data from the named variables into the buffer. The UPDATE statement writes the record back into file MOV.DAT. The CLOSE statement closes the file.

13.6 File and Record Operations

You can perform a variety of operations on files and on the records within a file. The following is a list of all the file and record operations supported by BASIC:

- Open a file for processing with the OPEN statement.
- Locate a record in a file with the FIND statement.
- Read a record from a file with the GET statement.
- Write a record to a file with the PUT statement.
- Delete a record from a file with the DELETE statement.
- Change the contents of a record field with the UPDATE statement.
- Unlock the last record accessed with the UNLOCK statement.
- Unlock all previously locked records with the FREE statement.
- Write data to a terminal-format file with the PRINT # statement.
- Reset the current record pointer to the beginning of a file with the RESTORE # and RESET # statements.
- Delete all the records after a certain point; that is, truncate the records, with the SCRATCH statement.
- Rename a file with the NAME AS statement.
- Close an open file with the CLOSE statement.
- Delete an entire file with the KILL statement.

Note that before you can perform any operations on the records in a file, you must first open the file for processing.

13.6.1 Opening Files

The OPEN statement opens a file for processing, specifies the characteristics of the file to RMS, and verifies the result. Opening a file with the specification FOR INPUT specifies that you want to use an existing file. Opening a file with the specification FOR OUTPUT indicates that you want to create a new file. If you do not specify FOR INPUT or FOR OUTPUT, BASIC tries to open an existing file. If no such file exists, a new file is created.

Clauses to the OPEN statement allow you to specify the characteristics of a file. All OPEN statement clauses concerning file or record format are optional when you open an existing file; those attributes that are not specified default to the attributes of the existing file. When you open an existing file, you must specify the file name, channel number, and unless the file is a terminal-format file, an organization clause. If you do not know the organization of the file you want to open, you can specify ORGANIZATION UNDEFINED. If you specify ORGANIZATION UNDEFINED, also specify RECORDTYPE ANY.

If you do not specify a map in the OPEN statement, the size of your program's record buffer is determined by the OPEN statement RECORDSIZE clause, or by the record size associated with the file. If you specify both a MAP clause and a RECORDSIZE clause in the OPEN statement, the specified record size overrides the size specified by the MAP clause.

The following statement opens a new sequential file of stream format records:

```
OPEN "TEST.DAT" FOR OUTPUT AS FILE #1%,        &
      SEQUENTIAL STREAM
```

The following example creates a relative file and associates it with a static record buffer. The MAP statement 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, and one double-precision, floating-point number. The record size is the total of these fields, or 28 bytes. All subsequent record operations use this static buffer for I/O to the file.

```
MAP (Inv_item) LONG Part_number,                        &
           STRING Inv_name = 16,                        &
           DOUBLE Unit_price
OPEN "INVENTORY.DAT" FOR OUTPUT AS FILE #1%             &
           ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY  &
           ,ALLOW READ, MAP Inv_item
```

The following OPEN statement opens a sequential file for reading only (ACCESS READ). Because the OPEN statement does not contain a MAP clause, a record buffer is created. This record buffer is 100 bytes long.

```
OPEN "CASE.DAT" AS FILE #1%                    &
          ,ORGANIZATION SEQUENTIAL VARIABLE    &
          ,ACCESS READ                         &
          ,RECORDSIZE 100%
```

When you do not specify a MAP statement, your program must use MOVE TO and MOVE FROM statements to move data between the record buffer and a list of variables.

The OPEN statement for indexed files must have a MAP clause. Moreover, if you are creating an indexed file, a PRIMARY KEY clause is required. You can create a segmented 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.

In the following example, the primary key is made up of three string variables. This key causes the records to be sorted in alphabetical order according to the user's last name, first name, and middle initial.

```
MAP (Segkey) STRING First_name = 15, MI = 1, Last_name = 15
OPEN "NAMES.IND" FOR OUTPUT AS FILE #1%,                  &
       ORGANIZATION INDEXED,                              &
       PRIMARY KEY (Last_name, First_name, MI),           &
       MAP Segkey
```

Note that there are restrictions on the maximum record size allowed for various file and record formats. See the *OpenVMS Record Management Services Reference Manual* for more information.

You can use logical names to assign a mnemonic name to all or part of a complete file specification, including node, device, and directory. The advantage in using logical names is that programs do not depend on literal file specifications. You can define logical names from the following:

- From DCL command level with the ASSIGN or DEFINE command
- From within a program with the SYS$CRELMN system service

BASIC supports any valid logical name as part of a file specification.

A logical name specifies a 1- to 255-character name to be associated with the specified device or file specification. If the logical name specifies a device, you must end the logical name with a colon. The following example defines a logical name for a file specification:

```
$ ASSIGN DUA1:[SENDER]PAYROL.DAT PAYROLL_DATA
```

This example defines a logical name for a physical device:

```
$ ASSIGN DUA2: DISK2:
```

Once you define the logical name, you can reference that name in your program. For example:

```
OPEN "PAYROLL_DATA" FOR INPUT AS FILE #1%,  &
        ORGANIZATION SEQUENTIAL
OPEN "DISK2:[SORT_DATA] SORT.LIS" FOR OUTPUT AS FILE #2%, &
        SEQUENTIAL VARIABLE
```

These OPEN statements do not depend on the availability of DUA1: or DUA2: in order to work. If these devices are not available, you can redefine the logical names so that they specify other disk drives before running the program. In addition, you can redirect the entire file specification for PAYROLL_DATA to point to the test or production version of the data.

13.6.2 Creating Virtual Array Files

BASIC virtual arrays let you define arrays that reside on disk. You use them just as you would an ordinary array. 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.

The following DIM # statement dimensions a virtual array on channel #1. The OPEN statement opens a virtual file that contains the array. The last program line assigns a value to one array element.

```
DIM #1%, LONG Int_array(10,10,10)
   .
   .
   .
OPEN "VIRT.DAT" FOR OUTPUT AS FILE #1%, VIRTUAL
   .
   .
   .
Int_array(5,5,5) = 100%
```

Note that you cannot redimension virtual arrays with an executable DIM statement. See Chapter 6 for more information about virtual arrays.

13.6.3 Locating Records

The FIND statement locates a specified record and makes it the current record. Using the FIND statement to locate records can be faster than using a GET statement because the FIND statement does not transfer any data to the record buffer; therefore, it executes faster than a GET statement. However, if you are interested in the contents of a record, you must retrieve it with a GET operation.

The FIND statement sets the current record pointer to the record just found, making it the target for a GET, UPDATE, or DELETE statement. (Note that you must have write access to a record before issuing a PUT, UPDATE, or DELETE operation.) A sequential FIND operation searches records in the following order:

- Sequential files from beginning to end
- Relative files in ascending relative record or cell number order

- Indexed files in ascending or descending order, based on the current key of reference and the key's collating sequence

For sequential fixed-length and relative files, you can find a particular record by specifying a RECORD clause. This is called a random access FIND. You can also perform a random access FIND for indexed files by specifying a key of reference, a relational test, and a key value.

In the following example, the first FIND statement finds the first record whose key value either equals or follows SMITH in the key's collating sequence. The second FIND statement finds the first record whose key value follows JONES in the key's collating sequence. Each record found by the FIND statement becomes the current record. (Note that you can only have one current record at a time.)

```
MAP (Emp) STRING Emp_name, LONG Emp_number, SSN
OPEN "EMP.DAT" AS FILE #1%, INDEXED,                &
         ACCESS READ,                               &
         MAP Emp,                                   &
         PRIMARY KEY Emp_name
FIND #1%, KEY #0% NXEQ "SMITH"
FIND #1%, KEY #0% NX "JONES"
```

The string expression can contain fewer characters than the key of the record you want to find. However, 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:

```
FIND #5%, KEY #0% EQ "TOM     "
FIND #5%, KEY #0% EQ "TOM"
```

The first FIND statement locates a record whose primary key field equals "TOM ". The second FIND statement locates the first record whose primary key field begins with "TOM".

Table 13–1 displays the status of the current record and next record pointers after both a sequential and a random access FIND.

Table 13–1 Record Context After a FIND Operation

Record Access Mode	File Type	Current Record	Next Record
Sequential FIND	Sequential	Record found	Current record + 1
	Relative	Record found	Next existing record
	Indexed	Record found	Next record in current key order
Random access FIND	All	Record found	Unchanged

Note that a random access FIND operation locates the specified record and makes it the current record, but the next record pointer does not change.

You can specify an ALLOW clause to the FIND statement if you have opened the file with ACCESS MODIFY or ACCESS WRITE and have specified UNLOCK EXPLICIT. The ALLOW clause lets you control the type of lock that RMS puts on the records you access. ALLOW NONE specifies that no other users can access this record (this is the default). ALLOW READ lets other users read the record; however, they cannot perform UPDATE or DELETE operations to this record. ALLOW MODIFY specifies that other users can both read and write to this record. This means that other access streams can perform GET, DELETE, or UPDATE operations to the specified record.

You can also specify a WAIT clause to the FIND statement; this clause allows you to wait for a record to become available in the event that it is currently locked by another process. In addition, you can specify a REGARDLESS clause; this clause allows you to read a locked record. For more information about the WAIT and REGARDLESS clauses, see Section 13.6.9.

13.6.4 Reading Records

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 sequential, relative, and indexed files. You should not use GET statements on terminal-format files or virtual array files.

For sequential files, a sequential GET retrieves the next record in the file. For relative files, a sequential GET retrieves the next existing record. For indexed files, a sequential GET retrieves the record with the next ascending or descending value in the current key of reference, depending on that key's collating sequence.

Table 13–2 shows the current record and next record pointers after a GET operation. Note that the values of these pointers vary, depending on whether or not the previous operation was a FIND.

Table 13–2 Record Context After a GET Operation

Record Access Mode	File Type	Current Record	Next Record
Sequential GET with FIND	Sequential	Record found	Current record + 1
	Relative	Record found	Next existing record
	Indexed	Record found	Next record in current key
Sequential GET without FIND	Sequential	Next record	Next record + 1
	Relative	Next existing record	Next existing record + 1
	Indexed	Next record in current key	Record following next record in current key
Random GET	All	Record specified	Next record in succession

If you precede a sequential GET operation with a FIND operation, the current record is the one located by FIND. If you do not perform a FIND operation before a sequential GET operation, the current record is the next sequential record.

The following example shows the use of the GET operation to sequentially access records in an indexed file. The example opens an indexed file and displays the first 25 records with serial numbers greater than AB2721 in ascending primary key value order.

```
MAP (Bec) STRING Owner = 30%, LONG Vehicle_number,     &
          STRING Serial_number = 22%
OPEN "VEH.IDN" FOR INPUT AS FILE #2%,                  &
      ORGANIZATION INDEXED, PRIMARY KEY Serial_number, &
      MAP Bec, ACCESS READ
GET #2%, KEY #0% EQ "AB2721"
FOR I% = 1% TO 25%
      GET #2%
      PRINT "Vehicle Number = ";Vehicle_number
      PRINT "Owner is: ";Owner
      PRINT
NEXT I%
```

The following example performs random GET operations by specifying a record number:

```
MAP (Bec) STRING Owner = 30%, LONG Vehicle_number,    &
          STRING Serial_number = 22%
OPEN "VEH.IDN" FOR INPUT AS FILE #2%,                 &
      ORGANIZATION SEQUENTIAL FIXED,                  &
      MAP Bec, ACCESS READ
INPUT "Which record do you want";A%
WHILE (A% <> 0%)
   GET #2%, RECORD A%
   PRINT "The vehicle number is", Vehicle_number
   PRINT "The serial number is", Serial_number
   PRINT "The owner of vehicle";Vehicle_number; "is", Owner
   INPUT "Next Record";A%
NEXT
CLOSE #2%
END
```

You can specify an ALLOW clause in a GET statement if you have opened the file with ACCESS MODIFY or ACCESS WRITE and UNLOCK EXPLICIT. The ALLOW clause lets you control the type of lock RMS places on the retrieved record. ALLOW NONE specifies that no other users can access this record (this is the default). ALLOW READ lets other access streams have read access to the record. That is, other users can retrieve the record, but cannot perform DELETE, PUT, or UPDATE operations on it. ALLOW MODIFY lets other access streams perform GET, DELETE, or UPDATE operations on the record.

If you are trying to access a locked record, BASIC signals “Record/bucket locked” (ERR=154). However, if you only need to read this record, you can override the lock with the REGARDLESS clause. The REGARDLESS clause allows you to read a locked record. Use caution when using the REGARDLESS clause because a record accessed in this way might be in the process of being changed by another program.

Alternatively, you can also specify the WAIT clause on a GET statement; the WAIT clause allows you to handle record locked conditions by waiting for the record to become available. Note that if a WAIT clause is specified on a GET operation to a unit-record device such as a terminal, the integer expression indicates how long to wait for the I/O to complete, rather than how long to wait on a record locked condition. For more information, see Section 13.6.9.

13.6.5 Writing Records

For a file opened with ACCESS WRITE or ACCESS MODIFY, the PUT statement moves data from the record buffer to a file using the I/O buffer. PUT statements are valid on RMS sequential, relative, and indexed files. You cannot use PUT statements on terminal-format files or virtual array files.

Sequential access is valid on RMS sequential, relative, and indexed files. For sequential, variable, and stream files, a sequential PUT operation adds a record at the end of the file. For sequential fixed and relative files, PUT writes records sequentially or randomly depending on the presence of a RECORD clause. For indexed files, RMS stores records in order of the primary key's collating sequence; therefore, you do not need to specify a random or sequential PUT. Table 13–3 shows the record context after both random and sequential PUT operations.

Table 13–3 Record Context After a PUT Operation

Record Access Mode	File Type	Current Record	Next Record
Sequential PUT	Sequential	None	End of file
Sequential PUT	Relative	None	Next record
Sequential PUT	Indexed	None	Undefined
Random PUT	Relative	None	Unchanged

After a PUT operation, the current record pointer has no value. However, the value of the next record pointer changes depending on the file type and the record access mode used with the PUT operation. In a sequential, stream, or variable file, records can only be added at the end of the file; therefore, the next record after PUT is the end of the file. In a relative, sequential, or fixed file, the next record after a PUT operation is the next logical record.

The following example opens a sequential file with ACCESS APPEND specified. For sequential files, this is almost identical to ACCESS WRITE. The only difference is that, with ACCESS APPEND, BASIC positions the file pointer after the last record in the file when it opens the file for processing. All subsequent PUT operations append the new record to the end of the existing file.

```
MAP (Buff) STRING Code = 4%, Exp_date = 9%, Type_desig = 32%
OPEN "INV.DAT"FOR INPUT AS FILE #2%,                  &
      ORGANIZATION SEQUENTIAL FIXED, ACCESS APPEND,   &
      MAP Buff
WHILE -1%
      INPUT "What is the specification code";Code
      INPUT "What is the expiration date";Exp_date
      INPUT "What is the designator";Type_desig
      PUT #2%
NEXT
```

If the current record pointer is not at the end of the file when you attempt a sequential PUT operation to a sequential file, BASIC signals "Not at end of file" (ERR=149).

In the following example, the PUT statement writes records to an indexed file. In this case, the error message "Duplicate key detected" (ERR=134) indicates that a record with a matching key field already exists, and you did not allow duplicates on that key.

```
10      MAP (Purchase_rec) STRING R_num = 5,              &
                               Dept_name = 10,          &
                               Pur_dat = 9
20      OPEN "INFO.DAT"FOR OUTPUT AS FILE #2,               &
                ORGANIZATION INDEXED FIXED, ACCESS WRITE, &
                PRIMARY KEY R_num, MAP Purchase_rec
30      WHILE -1%
                INPUT "Requisition number";R_num
                INPUT "Department name";Dept_name
                INPUT "Date of purchase";Pur_dat
                PRINT
                PUT #2%
        NEXT

Requisition number? 2522A
Department name? COSMETICS
Date of purchase? 15-JUNE-1985

Requisition number? 2678D
Department name? AUTOMOTIVE
Date of purchase? 15-JUNE-1985

Requisition number? 4167C
Department name? AUTOMOTIVE
Date of purchase? 6-JANUARY-1985

Requisition number? 2522A
Department name? SPORTING GOODS
Date of purchase? 25-FEBRUARY-1985
```

```
%BAS-F-DUPKEYDET, Duplicate key detected
-BAS-I-ON_CHAFIL, on channel 2 for file USER$$DISK:[MAGNUS]INFO.DAT;8 at
user PC 0017E593
-BAS-O-FROLINMOD, from line 30 in module DUPLICATES
-RMS-F-DUP, duplicate key detected (DUP not set)
```

13.6.6 Deleting Records

The DELETE statement removes a record from a file that was opened with ACCESS MODIFY. After you have deleted a record you cannot retrieve it. DELETE works with relative and indexed files only.

A successful FIND or GET operation must precede the DELETE operation. These operations make the target record available for deletion. In the following example, the FIND statement locates record 67 in a relative file and the DELETE statement removes this record from the file. Because the cell itself is not deleted, you can use the PUT statement to write a record into that cell after deleting its contents.

```
FIND #1%, RECORD 67%
DELETE #1%
```

Note

There is no current record after a deletion. The next record pointer is unchanged.

13.6.7 Updating Records

The UPDATE statement writes a new record at the location indicated by the current record pointer. UPDATE is valid on RMS sequential, relative, and indexed files.

UPDATE operates on the current record, provided that you have write access to that record. In order to successfully update a variable-length record, you must know the exact size of the record you want to update. BASIC has access to this information after a successful GET operation. If you have not performed a successful GET operation on the variable-length record, then you must specify a COUNT clause in the UPDATE statement that contains the record size information.

If you are updating a variable length record, and the record that you want to write out is of different size than the record you retrieved, you must use a COUNT clause.

An UPDATE will fail with the exception “No current record” (ERR=131) if you have not previously established a current record with a successful GET or FIND. Therefore, when updating records you should include error trapping in your program to make sure all GET operations execute successfully.

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 established a current record through a GET or FIND operation. Note that COUNT defaults to the size of the current record if a GET was performed. If a FIND operation was used to locate the current record, then you must supply a COUNT value.

The following program searches to find a record in which the *L_name* field matches the specified *Search_name$*. Once this record is found and retrieved, the *Rm_num* field of that record is updated; the program then prompts for another *Search_name$*. If a match is not found, BASIC prints the message “No such record” and prompts the user for another *Search_name$*. The program ends when the user enters a null string for the *Search_name$* value.

```
20      MAP (AAA) STRING L_name = 60%, F_name = 20%, Rm_num = 8%
30      OPEN "STU.DAT"FOR INPUT AS FILE #9%,           &
               ORGANIZATION SEQUENTIAL FIXED, MAP AAA
50      INPUT "Last name";Search_name$
55      Search_name$ = EDIT$(Search_name$, -1%)
60      IF Search_name$ = ""
               THEN GOTO 32010
        END IF
65      RESTORE #9%
70      WHEN ERROR IN
75         GET #9% WHILE Search_name$ <> L_name
        USE
           IF ERR=11
               THEN
                   PRINT "No such record"
                   CONTINUE 50
               ELSE
                   EXIT HANDLER
           END IF
        END WHEN
80      INPUT "Room number";Rm_num
90      UPDATE #9%
100     GOTO 50
32010   CLOSE #9%
32030   PRINT "Update complete"
32767   END
```

Note

An UPDATE operation invalidates the value of the current record pointer. The next record pointer is unchanged.

When you update a record in a relative variable file, the new record can be larger or smaller than the record it replaces, provided that it is smaller than the maximum record size set for the file. When you update a record in an indexed variable file, the new record can also be larger or smaller than the record it replaces. The updated record:

- Can be no longer than the maximum record size, if specified
- Must include at least the primary key field

The following program updates a specified record on an indexed file:

```
MAP (UPD) STRING Enrdat = 8%, LONG Part_num, Sh_num, REAL Cost
OPEN "REC.ING"FOR INPUT AS FILE #8%,                    &
      INDEXED, MAP UPD, PRIMARY KEY Part_num
INPUT "Part number to update";A%
Loop1:
WHILE -1%
   GET #8%, KEY #0%, EQ A%
   INPUT "Revised Cost is";Cost
   UPDATE #8%
   INPUT "Next Record";A%
   IF A% = 0%
   THEN
        EXIT Loop1
   END IF
NEXT
CLOSE #8%
END
```

If the new record either omits one of the old record's alternate key fields or changes one of them, the OPEN statement must specify a CHANGES clause for that key field when the file is created. Otherwise, BASIC signals the error "Key not changeable" (ERR=130).

13.6.8 Controlling Record Access

When you open a file, BASIC allows you to specify how you will access the file and what types of access you will allow other running programs while you have the file open.

If you open a file for read access only (ACCESS READ), BASIC by default allows other programs to have unrestricted access to the file. You can restrict access with an ALLOW clause only if the file's security constraints allow you write access to the file.

BASIC by default prevents access by other programs to any file you open with ACCESS WRITE, ACCESS MODIFY, or ACCESS SCRATCH (sequential files only). This default action is equivalent to specifying the OPEN statement ALLOW NONE clause. To allow less restrictive access to the open file, specify ALLOW READ or ALLOW MODIFY.

When a file is open for read access only and you have not restricted access to other programs with ALLOW NONE, BASIC allows other programs to read any record in the file including records that your program is concurrently accessing. However, when you retrieve a record with the GET statement from a file you have opened with the intent to modify, BASIC normally restricts other programs from accessing that record. This restriction is called **locking**.

To allow other programs to access a record you have locked, you must release the lock on the record in one of the following ways:

- Retrieve another record on the same channel. Unless you have opened the file with the UNLOCK EXPLICIT clause (see the following discussion), this action will unlock the previous record.
- Explicitly unlock the record with the UNLOCK or FREE statement. The UNLOCK statement releases the current record. The FREE statement releases all records locked on a given channel.
- Perform an UPDATE operation on the record. An UPDATE statement causes the current record to be unlocked.
- Close the file.

In addition to the capability of restricting access through the OPEN statement ALLOW clause, BASIC allows programs to explicitly control record locking on each record that is retrieved. To use explicit record locking on a file, the OPEN statement must include an UNLOCK EXPLICIT clause. You may then optionally specify an ALLOW clause on the GET and FIND statements. The ALLOW clause on a GET or FIND statement specifies the type of access allowed by other programs to the record while you are accessing it. The following statement specifies that other programs may read but not modify the records you have locked:

```
GET #1, ALLOW READ
```

If you specify UNLOCK EXPLICIT when opening a file, all records that you retrieve remain locked until you explicitly unlock them with a FREE, UNLOCK, or CLOSE statement.

13.6.9 Gaining Access to Locked Records

If you are trying to access a record that is currently locked, one possible solution is to use the REGARDLESS clause on the GET or FIND statement. The REGARDLESS clause is useful when you are interested in having only read access to the specified record. Be aware, however, that using the REGARDLESS clause to read a locked record can lead to unexpected results because the record you read can be in the process of being changed by another program.

Another solution is to include a WAIT clause on the GET or FIND statement. Note that you cannot specify a WAIT clause and a REGARDLESS clause on the same statement line. By specifying the WAIT clause, you can tell RMS to wait for a locked record to become available. You can optionally specify an integer expression from 0 to 255 with the WAIT clause. This integer expression indicates the number of seconds RMS should wait for a locked record to become available. If the record does not become available within the specified number of seconds, RMS signals the error "Keyboard wait exhausted" (ERR=15).

If you do not specify an integer expression with the WAIT clause, RMS waits indefinitely for the record to become available. Once the record becomes available, RMS delivers the record to the program.

Note that a deadlock condition can occur when you cause RMS to wait indefinitely for a locked record. A deadlock condition occurs when two users simultaneously try to access locked records in each other's possession. When a deadlock occurs, RMS signals the error, "RMS$_DEADLOCK". In turn, HP BASIC signals the error, "Detected deadlock error while waiting for GET or FIND" (ERR=193). To handle this error, you can either stop trying to access the particular record, or, if you must access the record, free all locked records (regardless of the channel) and then attempt the GET or FIND again. You need to unlock all records because you cannot know which record the other process wants.

Note

If the timeout value specified in the WAIT clause is less than the SYSGEN parameter DEADLOCK_WAIT, then a "Keyboard wait exhausted" (ERR=15) message can indicate that either the record did not become available during the specified time, or there is an actual deadlock situation. However, if the timeout value is greater than the SYSGEN parameter DEADLOCK_WAIT, the system correctly specifies that a deadlock situation has occurred.

The following example uses the WAIT clause to overcome a record locked condition and traps the resulting error condition:

```
MAP (worker) STRING first_name = 10,   &
                    last_name = 20,    &
                    badge_number = 6,  &
             LONG   dept_number

MAP (departments) STRING dept_name = 10,  &
                  LONG   dept_code
OPEN "Employee_data.dat" FOR INPUT AS FILE #1%,  &
      INDEXED FIXED, MAP worker, ACCESS MODIFY,  &
      PRIMARY badge_number

OPEN "departments.dat" FOR INPUT AS FILE #2,         &
     INDEXED FIXED, MAP departments, ACCESS MODIFY,  &
     PRIMARY dept_code

WHEN ERROR IN
   WHILE -1%
     GET #1, WAIT
     WHEN ERROR USE time_expired_handler
       GET #2%, KEY #0 EQ dept_number,  &
           WAIT 10%
     END WHEN
     PRINT badge_number, dept_name
   NEXT
USE
  SELECT ERR
    CASE = 11%
       PRINT "End of file reached"
       CLOSE 1%, 2%
    CASE = 193%
       PRINT "Deadlock detected"
       UNLOCK #2%
       RETRY
    CASE ELSE
       EXIT HANDLER
  END SELECT
END WHEN
```

```
HANDLER time_expired_handler
  IF ERR = 15% OR ERR = 193%
    THEN
       PRINT "Department info not available for:"
       PRINT "Employee ";badge_number
       PRINT "Going on to next record."
       CONTINUE
    ELSE
       EXIT HANDLER
  END IF
END HANDLER
END PROGRAM
```

The first WHEN ERROR block traps any deadlock conditions. The WHEN ERROR handler unlocks the current record on channel #2 in case another program is trying to access it and then retries the operation. The detached handler for the second WHEN ERROR block traps timeout errors and deadlock errors. If the desired information does not become available in the specified amount of time, or a deadlock condition occurs, the employee's badge number is printed out with an appropriate message, and the GET statement tries to retrieve the next record in the sequence.

13.6.10 Accessing Records by Record File Address

A Record File Address (RFA) uniquely specifies a record in a file. Accessing records by RFA is therefore more efficient and faster than other forms of random record access.[1]

Because an RFA requires six bytes of storage, BASIC has a special data type, RFA, that denotes variables that contain RFA information. Variables of data type RFA can be used only with the I/O statements and functions that use RFA information, and in comparison and assignment statements. You cannot print these variables or use them in any arithmetic operation. However, you can compare RFA variables using the equal to (=) and not equal to (<>) relational operators.

You cannot create named constants of the RFA data type. However, you can assign values from one RFA variable to another, and you can use RFA variables as parameters.

Accessing a record by RFA requires the following steps:

1. Explicitly declare the variable or array of data type RFA to hold the address.

[1] Record File Addresses do not exist for terminal-format files.

2. Assign the address to the variable or array element. You can do this either with the GETRFA function, or by reading a file of RFAs generated by previous GETRFA functions or by the VMS Sort Utility.
3. Specify 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. Therefore, 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.

The following example declares an array of type RFA containing 100 elements. After each PUT operation, the RFA of the record is assigned to an element of the array. Once the RFA 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.

```
DECLARE RFA R_array(1 TO 100)
DECLARE LONG I
MAP (XYZ) STRING A = 80
OPEN "TEST.DAT" FOR OUTPUT AS FILE #1,     &
      SEQUENTIAL, MAP XYZ
FOR I = 1% TO 100%
   .
   .
   .
    PUT #1
    R_array(I) = GETRFA(1%)
NEXT I
```

You can use the RFA clause on GET or FIND statements for any file organization; the only restriction is that the file must reside on a disk that is accessible to the node that is executing the program. An RFA value is only valid for the life of a specific version of a file. If a new version of a file is created, the RFA values might change. If you attempt to access a record with an invalid RFA value, HP BASIC signals a run-time error.

The following example continues the previous one. It randomly retrieves the records in a sequential file by using RFAs stored in the array.

```
DECLARE RFA R_array(1% TO 100%)
DECLARE LONG I
MAP (XYZ) STRING A = 80
OPEN "TEST.DAT" FOR OUTPUT AS FILE #1,     &
      SEQUENTIAL, MAP XYZ
FOR I = 1% TO 100%
   .
   .
   .
    PUT #1
    R_array(I) = GETRFA(1%)
NEXT I
WHILE -1%
    PRINT "Which record would you like to see";
    INPUT "(type a carriage return to exit)";Rec_num%
    EXIT PROGRAM IF Rec_num% = 0%
    GET #1, RFA R_array(Rec_num%)
    PRINT A
NEXT
```

13.6.11 Transferring Data to Terminal-Format Files

The PRINT # statement transfers program data to a terminal-format file. In the following example, the INPUT statements prompt the user for three values: *S_name$*, *Area$*, and *Quantity%*. Once these values are entered, the PRINT # statement writes these values to a terminal-format file that is open on channel #4.

```
FOR I% = 1% TO 10%
    INPUT "Name of salesperson":S_name$
    INPUT "Sales district";Area$
    INPUT "Quantity sold";Quantity%
    PRINT #4%, S_name$, Area$, Quantity%
NEXT I%
```

If you do not specify an output list in the PRINT # statement, a blank line is written to the terminal-format file. A PRINT statement without a channel number transfers program data to a terminal. See Chapter 5 for more information.

13.6.12 Resetting the File Position

The RESTORE # statement resets the current record pointer to the beginning of the file; it does not change the file. RESET # is a synonym for RESTORE. For example:

```
RESTORE #3%, KEY #2%
RESET #3%
```

The RESTORE # statement restores the file in terms of the second alternate key. The RESET # statement restores the file in terms of the primary key.

The RESTORE # statement can be used by all RMS file organizations. RESTORE without a channel number resets the data pointer for READ and DATA statements but does not affect any files.

13.6.13 Truncating Files

The SCRATCH statement is valid only on sequential files. Although you cannot delete individual records from a sequential file, you can delete all records starting with the current record through to the end of the file. In order to do this, you must first specify ACCESS SCRATCH when you open the file.

To truncate the file, locate the first record to be deleted. Once the current record pointer points to this record, execute the SCRATCH statement. The following program locates the thirty-third record and truncates the file beginning with that record.

```
OPEN "MMM.DAT" AS FILE #2%,            &
      SEQUENTIAL FIXED, ACCESS SCRATCH

first_bad_record = 33%

FIND #2%, RECORD first_bad_record
SCRATCH #2%
CLOSE #2%
END
```

SCRATCH does not change the physical size of the file; it reduces the amount of information contained in the file. (You can use the DCL command SET FILE/TRUNCATE to truncate the excess file space.) Therefore, you can write records with the PUT statement immediately after a SCRATCH operation.

13.6.14 Renaming Files

If the security constraints permit, you can change the name or directory of a file with the NAME...AS statement. For example:

```
NAME "MONEY.DAT" AS "ACCOUNTS.DAT"
```

This statement changes the name of the file MONEY.DAT to ACCOUNTS.DAT.

Note

The NAME...AS statement can change only the name and directory of a file; it cannot be used to change the device name.

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.

13.6.15 Closing Files and Ending I/O

All programs should close files before the program terminates. However, files are automatically closed in the following situations:

- At an END, END PROGRAM, or EXIT PROGRAM statement
- When it completes the last statement in the program if no END statement exists
- While executing a CHAIN statement

Files are not closed after executing a STOP, END SUB, END FUNCTION, or END PICTURE statement.

The CLOSE statement closes files and disassociates these files and their buffers from the channel numbers. If the file is a magnetic tape device and the data is written to a tape, CLOSE writes trailer labels at the end of the file. The following is an example of the CLOSE statement:

```
CLOSE #1%
B% = 4%
CLOSE #2%, B%, 7%
CLOSE I% FOR I% = 1% TO 20%
```

13.6.16 Deleting Files

If the security constraints permit, you can delete a file with the KILL statement. For example:

```
KILL "TEST.DAT"
```

This statement deletes the file named TEST.DAT. Note that this statement deletes only the most current version of the file. Do not omit the file type, because there is no default. You can delete only one file at a time; to delete all versions of a file matching a file specification, use the Run-Time Library routine LIB$DELETE_FILE.

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 once you have deleted it.

13.7 File-Related Functions

The following built-in functions are provided for finding:

- The characteristics of the last file opened (FSP$)
- The number of bytes moved in the last I/O operation (RECOUNT)
- The file status (STATUS, VMSSTATUS, and RMSSTATUS)

These functions are discussed in the following sections.

13.7.1 FSP$ Function

If you do not know the organization of a file, you can find out by opening the file for input with the ORGANIZATION UNDEFINED and RECORDTYPE ANY clauses. 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. For example:

```
RECORD FSP_data
       VARIANT
       CASE
          BYTE Org
          BYTE Rat
          WORD Max_record_size
          LONG File_size
          WORD Bucketsize_blocksize
          WORD Num_keys
          LONG Max_record_number
       CASE
          STRING Ret_string = 16
       END VARIANT
END RECORD

DECLARE FSP_data File_chars

OPEN "FIL.DAT" FOR INPUT AS FILE #1%,              &
    ORGANIZATION UNDEFINED,                        &
    RECORDTYPE ANY, ACCESS READ
File_chars::Ret_string = FSP$(1%)
```

The following list explains the above example:

- *Rat* returns the low byte that is the RMS record attributes (RAT) field.
- *Org* returns the high byte that is the RMS organization (ORG) field.
- *Max_record_size* returns the RMS maximum record size (MRS) field.
- *File_size* returns the RMS allocation quantity (ALQ) field.
- *Bucketsize_blocksize* 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.
- *Max_record_number* returns the RMS maximum record number (MRN) field if the file is a relative file.

Note that FSP$ returns zeros in bytes 9 to 12. For more information, see the *OpenVMS Record Management Services Reference Manual*.

13.7.2 RECOUNT Function

Read operations can transfer varying amounts of data. The system variable RECOUNT contains the number of characters (bytes) read after each read operation.

After a read operation from your terminal, RECOUNT contains the number of characters transferred, including the line terminator. After accessing a record, RECOUNT contains the number of characters in the record.

RECOUNT is reset by every read operation on any channel, including the controlling terminal. Therefore, if you need to use the value of RECOUNT, copy it to another variable before executing another read operation. RECOUNT is undefined if an error occurs during a read operation.

RECOUNT is often used as the argument to the COUNT clause in the UPDATE or PUT statement for variable-length files. The following sequence of statements ensures that the output record on channel #5 is the same length as the input record on channel #4:

```
GET #4%
bytes_read% =  RECOUNT
   .
   .
   .
PUT #5%, COUNT bytes_read%
```

13.7.3 STATUS, VMSSTATUS, and RMSSTATUS Functions

The STATUS function accesses the status longword that contains characteristics of the last opened file. If an error occurs during an input operation, the value of STATUS is undefined. If an error does not occur, the six low-order bits of the returned value contain information about the type of device accessed by the last input operation. These bits correspond to the following devices:

- If bit 0 is set, the device type is a record-oriented device.
- If bit 1 is set, the device type is a carriage control device.
- If bit 2 is set, the device type is a terminal.
- If bit 3 is set, the device type is a directory oriented device.
- If bit 4 is set, the device type is a single directory device.
- If bit 5 is set, the device type is a sequential block-oriented device (magnetic tape or TK50).

Both the VMSSTATUS and RMSSTATUS functions are used to determine which non-BASIC error caused a resulting BASIC error. In particular, VMSSTATUS can be used for any non-BASIC errors, while RMSSTATUS is used specifically for RMS errors. For more information about these functions, see Chapter 15 and the *HP BASIC for OpenVMS Reference Manual*.

13.8 OPEN Statement Options

This section explains the OPEN statement keywords that enable you to control how a file is created or opened. These keywords are:

BUCKETSIZE
BUFFER
CONNECT
CONTIGUOUS
DEFAULTNAME
EXTENDSIZE
FILESIZE
NOSPAN
RECORDTYPE
TEMPORARY
USEROPEN
WINDOWSIZE

13.8.1 BUCKETSIZE Clause

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 consists of one or more disk blocks. The default bucket size is the record size rounded up to a block boundary. Although RMS defines the bucket size in terms of disk blocks, the BUCKETSIZE clause specifies the number of records a bucket contains. For example:

```
OPEN  "STOCK_DATA.DAT" FOR OUTPUT AS FILE #1%,          &
      ORGANIZATION RELATIVE FIXED, BUCKETSIZE 12%
```

This example specifies a bucket containing approximately 12 records. RMS reads in entire buckets into the I/O buffer at once, and a GET statement transfers one record from the I/O buffer to your program's record buffer.

When you open an existing relative or indexed file and specify a bucket size other than that originally assigned to the file, BASIC signals the error, "File attributes not matched" (ERR=160).

Records cannot span bucket boundaries. Therefore, when you specify a bucket size in your program, you must consider the size of the largest record in the file. Note that a bucket must contain at least one record. Buckets in both relative and indexed files contain information in addition to the records stored in the bucket. You should take this into consideration.

There are two ways to establish the number of blocks in a bucket. The first is to use the default. The second is to specify the approximate number of records you want in each bucket. A bucket size based on that number is then calculated.

The default bucket size assigned to relative and indexed files is as small as possible. A small bucket size, however, is rarely desirable.

A default bucket size is selected depending on the:

- Record length
- File organization (relative or indexed)
- Record format

If you do not define the BUCKETSIZE clause in the OPEN statement, BASIC does the following:

- Assumes that there is a minimum of one record in the bucket
- Calculates a size
- Assigns the appropriate number of blocks

Note that when you specify a bucket size for files in your program, you must keep in mind the space versus speed tradeoffs. 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 and the processing time required to search the bucket. Conversely, a small bucket size minimizes buffer requirements, but increases the number of accesses to the storage device, thereby decreasing the speed of operations.

It is recommended that you use the DCL command EDIT/FDL to design files used in production applications where performance is a concern.

13.8.2 BUFFER Clause

The BUFFER clause applies to disk files of any organization. In the case of sequential files, the BUFFER clause sets the number of blocks read in on each disk access. For relative and indexed files, the BUFFER clause determines the number of I/O buffers that are allocated. In general, the OpenVMS operating system supplies adequate defaults for all file types; therefore, the BUFFER clause is rarely necessary.

You can specify up to 127 buffers as either a positive or a negative number:

- If (0 < BUFFER < 127), RMS allocates enough space for the specified number of buckets.
- If (-128 < BUFFER < 0), BASIC allocates the absolute value of the specified number of buffers.
- If (BUFFER=0), BASIC allocates the process default for the particular file organization and device—this value is usually adequate.

13.8.3 CONNECT Clause

The CONNECT clause can be used only on indexed files. CONNECT lets you process different groups of records on different indexed keys or on the same key without incurring all of 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.

```
MAP (Indmap) WORD Emp_num,                    &
     STRING Emp_last_name = 20,               &
     SINGLE Salary,                           &
     STRING Wage_code = 2
OPEN "IND.DAT" FOR INPUT AS FILE #1%,         &
     ORGANIZATION INDEXED,                    &
     MAP Indmap,                              &
     PRIMARY KEY Emp_num,                     &
     ALTERNATE KEY Emp_last_name
   .
   .
   .
OPEN "IND.DAT" FOR INPUT AS FILE #2%          &
     ORGANIZATION INDEXED,                    &
     MAP Indmap,                              &
     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; connected channels are called **children**. More than one OPEN statement can connect to the parent channel; however, you cannot connect to a channel that has already been connected to another channel.

Do not use the CONNECT clause when accessing files on remote DECnet nodes.

13.8.4 CONTIGUOUS Clause

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 does not need to 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. If there is not enough contiguous disk space, BASIC allocates as much contiguous space as possible. (For truly contiguous records, you must use the USEROPEN clause and set the CTG bit in the FAB FOP field—see the *OpenVMS Record Management Services Reference Manual*.)

Opening a file with both the FILESIZE and CONTIGUOUS clauses pre-extends the file contiguously or in as few disk extents as possible.

13.8.5 DEFAULTNAME Clause

The DEFAULTNAME clause in the OPEN statement lets you specify a default file specification for the file to be opened. It is valid with all file organizations. BASIC uses the DEFAULTNAME clause for any part of the file specification that is not explicitly supplied.

```
LINPUT "Next data file";Fil$
OPEN Fil$ FOR INPUT AS FILE #5%,       &
     ORGANIZATION SEQUENTIAL,          &
     DEFAULTNAME "USER$DEVICE:.DAT"
```

The DEFAULTNAME clause supplies default values for the device, directory, and file type portions of the file specification. Typing ABC in response to the Next data file? prompt causes BASIC to try to open USER$DEVICE:ABC.DAT.

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 SYS$DEVICE:ABC in response to the prompt, BASIC tries to open SYS$DEVICE:ABC.DAT. In this case, SYS$DEVICE: overrides the device default in the DEFAULTNAME clause. Any part of the file specification still missing is filled in from the current default device and directory of the process.

13.8.6 EXTENDSIZE Clause

The EXTENDSIZE attribute determines how many disk blocks RMS adds to the file when the current allocation is exhausted. The EXTENDSIZE clause only has an effect when creating a file. You specify EXTENDSIZE as a number of blocks. For example:

```
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 must conform to the following requirements:

- It must be specified when you create the file
- It cannot exceed 65,535 disk blocks

If you specify zero, the extension size equals the RMS default value. The EXTENDSIZE value can be overridden for single OPEN operations.

13.8.7 FILESIZE Clause

With the FILESIZE attribute, you can allocate disk space for a file when you create it. The following statement allocates 50 blocks of disk space for the file VALUES.DAT:

```
OPEN "VALUES.DAT" FOR OUTPUT AS FILE #3%, FILESIZE 50%
```

Pre-extending a file has several advantages:

- The system can create a complete directory structure for the file, instead of allocating and mapping additional disk blocks when needed.
- You reserve the needed disk space for your application. This ensures that you do not run out of space when the program is running.
- Pre-extension can make some of the file's disk blocks contiguous, especially when used with the CONTIGUOUS keyword.

Note that pre-extension can be a disadvantage if it allocates disk space needed by other users. The FILESIZE clause is ignored when HP BASIC opens an existing file.

13.8.8 NOSPAN Clause

By default, sequential files allow records to cross or span block boundaries. If records cross block boundaries, RMS packs records into the file end-to-end throughout the file, leaving space for control information and padding.

The NOSPAN clause overrides this default, forcing records to fit into individual blocks (with space provided for control information and padding). 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 each block. However, when records span block boundaries, RMS writes records end-to-end without regard for block boundaries. For example, if you specify 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 writing that record in the next block. This minimizes wasted disk space and improves the file's capacity, at the minimal expense of increased processing overhead.

13.8.9 RECORDTYPE Clause

The RECORDTYPE clause lets you specify record formats that are compatible with files created by other language processors. You can choose one of four qualifiers: LIST, FORTRAN, ANY, and 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 RMS when writing the file to a unit-record device.

If your program accesses a file created with a Fortran language processor, use the FORTRAN qualifier. In the following example, the FORTRAN qualifier sets the FORTRAN carriage control attribute in the RAT field in the FAB. For more information about the FAB control structure, see Section 13.8.11. The first byte of the record is assumed to be the carriage control information. For example:

```
OPEN "FIL.DAT" FOR INPUT AS FILE #1%,      &
     ORGANIZATION SEQUENTIAL, RECORDTYPE FORTRAN
```

If your program accesses a file created by an unknown language processor or by DCL, use the ANY qualifier; this qualifier causes BASIC to handle any record attribute type. If you create a file with the ANY qualifier, BASIC uses the default of LIST. For example:

```
OPEN "FIL.DAT" FOR INPUT AS FILE #1%,       &
     ORGANIZATION INDEXED, RECORDTYPE ANY
```

13.8.10 TEMPORARY Clause

If you specify the TEMPORARY clause in the OPEN statement, BASIC deletes that file in any one of the following cases:

- When you close the file
- When the program aborts or exits
- When your process terminates

No entry for this file is made in any directory.

13.8.11 USEROPEN Clause

The USEROPEN clause specifies an external long function that BASIC executes when you open or create a file. (You do not need to declare the USEROPEN routine with an EXTERNAL FUNCTION statement.) This procedure can then specify additional OPEN parameters for the file. For example:

```
OPEN "FILE.DAT" FOR INPUT AS FILE #2%,      &
     ORGANIZATION INDEXED, USEROPEN Myopen, MAP ABC
```

The code in *Myopen* determines how the file FILE.DAT is opened. The Run-Time Library sets up six RMS control structures before calling the USEROPEN procedure. Table 13–4 defines these structures and their meanings.

Table 13–4 RMS Control Structures Set for the USEROPEN Clause

Structure	Definition
FAB	File Access Block
RAB	Record Access Block
NAM	Name Block
XAB	FHC Extended Attributes Block
ESA	Expanded Name String
RSA	Resultant Name String

A USEROPEN procedure should not alter the allocation of these structures, although it can modify the contents of many of the fields. You should not modify fields set by other OPEN statement keywords. For example, you should use the RECORDSIZE clause, not a USEROPEN routine, to set the record length.

The allocation of the RMS control structures (except for the RAB) lasts only for the duration of the OPEN statement. Therefore, your USEROPEN can retain only the RAB address for use after the OPEN operation is complete. Note that any additional structures that you allocate and link into the RMS structures must be unlinked before exiting the USEROPEN.

Note

Future releases of the OpenVMS Run-Time Library might alter the use of some RMS fields. Therefore, you might have to alter your USEROPEN procedures accordingly.

The following steps describe the execution of the USEROPEN routine:

1. BASIC performs normal OPEN statement processing up to the point where it would call the RMS OPEN/CREATE and CONNECT routines. BASIC then passes control to the USEROPEN routine.
2. BASIC passes the address of the FAB as the first parameter, the address of the RAB as the second parameter, and the address of the user-specified channel number as the third parameter to the routine.
3. The USEROPEN routine can modify the contents of the RMS control structures, and it must call the RMS OPEN or RMS CREATE routine and the RMS CONNECT routine and return the status in R0.

Example 13–1 shows how to create a USEROPEN routine to obtain a RAB address.

Example 13–1 Creating a USEROPEN Routine

```
%TITLE  "Example USEROPEN"
%SBTTL  "Useropen Routine to obtain RAB address"
%IDENT  "Version 1.1"

FUNCTION LONG Get_rab_address ( Fabdef User_fab, Rabdef User_rab,
                                LONG Channel )
!++
! FUNCTIONAL DESCRIPTION:
!
!   Save the address of the RMS Record Access Block allocated by the caller
!   in a global symbol.  Open the file and return the status from RMS.
!
! FORMAL PARAMETERS (Standard for all BASIC USEROPEN procedures)
!
!   User_fab    Address of RMS File Access Block
!   User_rab    Address of RMS Record Access Block
!   Channel     Logical Unit assigned to file by caller.
!
! RETURN VALUE:    RMS Status value
!
! GLOBAL COMMON USAGE
!
!   RAB_ptr     Single longword PSECT used to pass RAB address to caller.
!
!--
    OPTION  INACTIVE = SETUP,           &
         CONSTANT TYPE = INTEGER,       &
         TYPE = EXPLICIT

    %NOLIST
    %INCLUDE "$FABDEF" %FROM %LIBRARY "SYS$LIBRARY:BASIC$STARLET"
    %INCLUDE "$RABDEF" %FROM %LIBRARY "SYS$LIBRARY:BASIC$STARLET"
    %INCLUDE "$RMSDEF" %FROM %LIBRARY "SYS$LIBRARY:BASIC$STARLET"
    %INCLUDE "STARLET" %FROM %LIBRARY "SYS$LIBRARY:BASIC$STARLET"
    %LIST
    !+
    !   Common area used to pass RAB address to caller.
    !-
    COMMON     (RAB_ptr)    LONG     rab_address

    DECLARE LONG    Rms_status
    !+
    !   Save RAB address in global symbol known to caller.
    !   Perform standard RMS open sequence
    !-
    Rab_address = LOC(User_rab::rab$b_bid)

    Rms_status = Sys$open( User_fab )
```

(continued on next page)

Example 13–1 (Cont.) Creating a USEROPEN Routine

```
    IF Rms_status = Rms$_normal
    THEN
        Rms_status = Sys$connect( User_rab )
    END IF

END FUNCTION Rms_status
```

Note

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; any values placed there by the USEROPEN routine are overwritten. Also, you must not set RMS Locate mode when using a USEROPEN routine on sequential files.

13.8.12 WINDOWSIZE Clause

The WINDOWSIZE clause specifies the number of block retrieval pointers in memory for the file. WINDOWSIZE is not a file attribute, and therefore can be changed any time you open a file.

Retrieval pointers are associated with the file header and point to contiguous blocks on disk. By keeping retrieval pointers in memory, you can reduce the I/O associated with locating a record because 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 is 7.

A value of zero specifies the default number of retrieval pointers. A value of –1 specifies mapping the entire file, if possible. Values from –128 to –2 are reserved.

14

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. This chapter describes how to format data with the PRINT USING statement.

14.1 Overview of the PRINT USING Statement

The ability to format data with the PRINT USING statement is useful because the way in which HP BASIC displays data with the PRINT statement is often limited. For example, a program might use floating-point numbers to represent dollars and cents. The PRINT statement displays floating-point numbers with up to six digits of accuracy, and places the decimal point anywhere in that 6-digit field. In contrast, PRINT USING lets you display floating-point numbers in the following ways:

- Rounded to a number of specified 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 provides a more readable report. Another use for formatted numeric values might be to print checks on a printer. 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.

It is recommended that you declare all format expressions as string constants. When you do this the HP BASIC compiler instructs the Run-Time Library to compile the string at compile time rather than at run time, thus improving the performance of your code.

14.2 Using Format Strings

Format strings determine the way in which items are to be printed in the output file. Format strings can be any of the following:

- String variables
- String literals
- Named string constants
- A combination of the previous strings

The PRINT USING statement must contain one or more format strings. Each format string is made up of one **format field**. Each format field controls the output of one print item and can contain only certain characters, as described throughout the chapter.

The PRINT USING statement must also contain a list of items you want printed. To format print items, you must separate them 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, HP 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.

When HP BASIC encounters an invalid character within the current format field, it automatically ends the format field; therefore, you do not need to delimit format fields. The character that terminates the previous field can be either a new format field or a string literal.

In the following example, 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, HP BASIC ends the first format field after the third character. HP BASIC continues to scan the format string, searching for a character that begins a format field. The first such character is the number sign at character position 7. Therefore, the characters at positions 4, 5, and 6 are treated as a string literal. The characters at positions 7, 8, and 9 make up a second valid numeric format field.

```
PRINT USING "###ABC###", 123, 345
```

Output

```
123ABC345
```

When the statement executes, HP 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. If you were to supply a third number in the list, HP BASIC would reuse the first format string. This is called **reversion**.

```
PRINT USING "###ABC###", 123, 345,
564
```

Output

```
123ABC345
564ABC
```

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 as shown in the following example:

```
DECLARE STRING CONSTANT format_string = "###.##  ###.##"
DECLARE SINGLE A,B
A = 2.565
B = 100.350
PRINT USING format_string, A, B, A, B
```

Output

```
  2.57  100.35
  2.57  100.35
```

When the HP BASIC compiler encounters the PRINT USING statement, HP BASIC prints the value of *A* (rounded according to PRINT USING rules), three spaces, then the value of *B*. HP BASIC prints the three spaces because they are treated as a string literal in the format string. Notice that when HP BASIC reuses a format string, it begins on a new line.

14.3 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-zero fields
- A special character that 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), HP BASIC prints a percent sign (%) to signal this condition and displays the number in PRINT format.

14.3.1 Specifying the Number of Digits

You reserve places for digits by including a number sign (#) for each digit position. If you print negative numbers, you must also reserve a place for the minus sign.

```
PRINT USING "###",123      !Three places reserved
PRINT USING "#####",12345  !Five places reserved
PRINT USING "####",-678    !Four places reserved
END
```

Output

```
123
12345
-678
```

If there are not enough digits to fill the field, HP BASIC prints spaces before the first digit.

```
format_string$ = "#####"
PRINT USING format_string$, 1
PRINT USING format_string$, 10
PRINT USING format_string$, -1709
PRINT USING format_string$, 12345
END
```

Output

```
     1
    10
 -1709
 12345
```

If you have not reserved enough digits to print the fractional part of a number, HP BASIC rounds the number to fit the field.

```
PRINT USING "###",126.7
PRINT USING "#",5.9
PRINT USING "#",5.4
END
```

Output

```
127
6
5
```

If you have not reserved enough places to print a number's integer portion, HP BASIC prints a percent sign as a warning followed by the number in PRINT statement format. After HP BASIC prints the number, it completes the rest of the list in PRINT USING format.

In the following example, PRINT USING displays the first number. Because there are not enough places to the left of the decimal point to display a 3-digit number, BASIC prints the second number in PRINT statement format, with a space before and after, but includes a percent sign warning.

```
PRINT USING "###", 256
PRINT USING "##", 256
END
```

Output

```
256
% 256
```

14.3.2 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 the remaining spaces with zeros.

```
DECLARE STRING CONSTANT FM = "##.###"
PRINT USING FM, 15.72
PRINT USING FM, 39.3758
PRINT USING FM, 26
```

Output

```
15.720
39.376
26.000
```

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 percent sign. The following example shows how PRINT USING rounds numbers when you specify decimal point location:

```
PRINT USING "##.##", 25.789
PRINT USING "##.###", 100.2
PRINT USING "#.##",.999
END
```

Output

```
25.79
% 100.2
1.00
```

BASIC fills all reserved spaces to the left of the decimal point with specified digits, spaces, or the minus sign.

```
PRINT USING "##.##", 5.25
PRINT USING "##.##", -5.25
PRINT USING "###.##,-5.25
END
```

Output

```
 5.25
-5.25
 -5.25
```

14.3.3 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 14–1 summarizes these special characters.

Table 14–1 Format Characters for Numeric Fields

Character	Effect on Format
Number sign (#)	Reserves a place for one digit.
Decimal point (period)(.)	Determines decimal point location and reserves a place for the radix point.
Comma (,)	Prints a comma before every third digit to the left of the decimal point and reserves a place for one digit or digit separator.
Two asterisks (**)	Print leading asterisks before the first digit and reserve places for two digits.

(continued on next page)

Table 14–1 (Cont.) Format Characters for Numeric Fields

Character	Effect on Format
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, see Section 14.3.3.3
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.
Zero in angle brackets (<0>)	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 in angle brackets (<CD>)	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.

14.3.3.1 Commas

You can place a comma anywhere in a number field to the left of the decimal point or to the right of the field's first character. A comma cannot start a format field. BASIC 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.

```
PRINT USING "##,###",10000
PRINT USING "##,###",759
PRINT USING "$$#,###.##",25694.3
PRINT USING "**#,###",7259
PRINT USING "####,#.##",25239
END
```

Output

```
 10,000
    759
 $25,694.30
 **7,259
 25,239.00
```

14.3.3.2 Asterisk-Fill Fields

To print an asterisk (*) before the first digit of a number, you must start the field with two asterisks.

```
DECLARE STRING CONSTANT FM = "**##.##"
PRINT USING FM, 1.2
PRINT USING FM, 27.95
PRINT USING FM, 107
PRINT USING FM, 1007.5
END
```

Output

```
***1.20
**27.95
*107.00
1007.50
```

Note that the asterisks reserve two places as well as cause asterisk fill.

To specify a negative number in an asterisk-fill field, you must place a trailing minus sign in the field. The trailing minus sign must be the last character in the format string.

```
DECLARE STRING CONSTANT FM = "**##.##-"
PRINT USING FM, 27.95
PRINT USING FM, -107
PRINT USING FM, -1007.5
END
```

Output

```
**27.95
*107.00-
1007.50-
```

If you try to print a negative number in an asterisk-fill field that does not include a trailing minus sign, BASIC signals “PRINT USING format error” (ERR=116).

You cannot specify both asterisk-fill and zero-fill for the same numeric field.

14.3.3.3 Currency Symbols

To print a currency symbol before the first digit of a number, you must start the field with two dollar signs. If the data contains both positive and negative numbers, you must include a trailing minus sign.

```
DECLARE STRING CONSTANT FM = "$$##.##-"
PRINT USING FM, 77.44
PRINT USING FM, 304.55
PRINT USING FM, 2211.42
PRINT USING FM, -125.6
PRINT USING FM, 127.82
END
```

Output

```
 $77.44
$304.55
% 2211.42
$125.60-
$127.82
```

Note that the dollar signs reserve places for the currency symbol and only one digit; the dollar sign is always printed. (Hence the warning indicator (%) when the third PRINT USING statement executes.) Contrast this with the asterisk-fill field, where BASIC prints asterisks only when there are leading spaces.

By default, the currency symbol is a dollar sign. On OpenVMS systems, you can change the currency symbol, radix point, and digit separator by assigning the characters you want to the logical names SYS$CURRENCY, SYS$RADIX_POINT, and SYS$DIGIT_SEP, respectively.

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 and DR formatting character, BASIC signals "PRINT USING Format error" (ERR=116).

14.3.3.4 Negative Fields

To allow for a field containing negative values, you must place a trailing minus sign in the format field. A negative format field causes the value to be printed with a trailing minus sign. You can also denote negative fields with CR and DR. See Section 14.3.3.8 for more information.

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 as shown in Example 1, and a space after positive numbers as shown in Example 2.

Example 1

```
!Standard field
PRINT USING "###.##",-10.54
PRINT USING "###.##",10.54
END
```

Output 1

```
-10.54
 10.54
```

Example 2

```
!Fields with Trailing Minus Signs
PRINT USING "##.##-",-10.54
PRINT USING "##.##-",10.54
END
```

Output 2

```
10.54-
10.54
```

14.3.3.5 E (Exponential) Format

To print a number in E format, you must 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)
- An exponent (the exponent is 2 digits for single, double, and s_floating, 3 digits for g_floating and t_floating, and 4 digits for h_floating and x_floating)

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.

```
PRINT USING "###.##^^^^",5
PRINT USING "###.##^^^^",1000
PRINT USING ".##^^^^",5
END
```

Output

```
 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.

```
PRINT USING "###.##^^^",5
PRINT USING "###.##^^^^^",5
END
```

Output

```
   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 prints a percent sign (%) as a warning.

You cannot use exponential format with asterisk-fill, floating-dollar sign, or trailing minus formats.

14.3.3.6 Leading Zeros

To print leading zeros in a numeric field, you must start the format field with a zero enclosed in angle brackets (<0>). These characters also reserve one place for a digit.

```
DECLARE STRING CONSTANT FM = "<0>####.##"
PRINT USING FM, 1.23, 12.34, 123.45, 1234.56, 12345.67
```

Output

```
00001.23
00012.34
00123.45
01234.56
12345.67
```

When you specify zero-fill, you cannot specify asterisk-fill or floating-dollar sign format for the same field.

14.3.3.7 Blank-If-Zero Fields

To print a blank field for values which round to zero, you must start the numeric field with a percent sign enclosed in angle brackets (<%>).

In the following example, 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.

```
DECLARE STRING CONSTANT FM = "<%>####.##"
PRINT USING FM, 1000, 0, .001, -5000
```

Output

```
  1000.00

 -5000.00
```

14.3.3.8 Debits and Credits

You can have BASIC use credit and debit notation to differentiate positive and negative numbers. To do this, you place the characters <CD> (Credit/Debit) at the end of the numeric format string. This causes BASIC to print CR (Credit Record) after negative numbers, and DR (Debit Record) after positive numbers and zero.

```
DECLARE STRING CONSTANT FM = "$$####.##<cd>"
PRINT USING FM, -552.35, 200, -5
```

Output

```
 $552.35CR
 $200.00DR
   $5.00CR
```

You cannot use a trailing minus sign and Credit/Debit formatting in the same numeric field. Using the Credit/Debit formatting character causes the value to be printed with a leading space.

14.4 Printing Strings

With the PRINT USING statement, you can specify the following aspects of string format:

- The number of characters
- Left-justified format
- Right-justified format
- Centered format
- Extended field format

Table 14–2 summarizes the format characters and their effects.

Table 14–2 Format Characters for String Fields

Character	Effect on Format
Single quotation mark (')	Starts the string field and reserves a place for one character.
L (upper- or lowercase)	Left-justifies the string and reserves a place for one character.
R (upper- or lowercase)	Right-justifies the string and reserves a place for one character.
C (upper- or lowercase)	Centers the string in the field and reserves a place for one character.
E (upper- or lowercase)	Left-justifies the string; expands the field, as necessary, to print the entire string; and reserves a 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. It is recommended that you 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. It is recommended that you not use this type of field for new program development. Instead, use a single quotation mark to create a 1-character field.

You must start string format fields 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. For more information about extended fields, see Section 14.4.4.

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.

```
PRINT USING "'","ABCDE"
END
```

Output

```
A
```

See Section 14.4.4 for an example of different types of fields used together.

14.4.1 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.

```
PRINT USING "'LLLLLL","ABCDE"
PRINT USING "'LLLL","ABC"
PRINT USING "'LLLLL","12345678"
END
```

Output

```
ABCDE
ABC
123456
```

14.4.2 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 quotation mark (') followed by a series of Rs.

```
DECLARE STRING CONSTANT right_justify = "'RRRRR"
PRINT USING right_justify,"ABCD"
PRINT USING right_justify,"A"
PRINT USING right_justify,"STUVWXYZ"
END
```

Output

```
  ABCD
     A
STUVWX
```

14.4.3 Centered Fields

BASIC prints strings in a centered field by aligning the center of the string with 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.

```
DECLARE STRING CONSTANT center = "'CCCC"
PRINT USING center, "A"
PRINT USING center, "AB"
PRINT USING center, "ABC"
PRINT USING center, "ABCD"
PRINT USING center, "ABCDE"
END
```

Output

```
   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.

14.4.4 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.

```
PRINT USING "'E", "THE QUICK BROWN"
PRINT USING "'EEEEEEE', "FOX"
END
```

Output

```
THE QUICK BROWN
FOX
```

The following example uses left-justified, right-justified, centered, and extended fields:

```
PRINT USING "'LLLLLLLLL","THIS TEXT"
PRINT USING "'LLLLLLLLLLLLLL","SHOULD PRINT"
PRINT USING "'LLLLLLLLLLLLLL",'AT LEFT MARGIN'
PRINT USING "'RRRR","1,2,3,4"
PRINT USING "'RRRR",'1,2,3'
PRINT USING "'RRRR','1,2"
PRINT USING "'RRRR","1"
PRINT USING "'CCCCCCCCC","A"
PRINT USING "'CCCCCCCCC","ABC"
PRINT USING "'CCCCCCCCC","ABCDE"
PRINT USING "'CCCCCCCCC","ABCDEFG"
PRINT USING "'CCCCCCCCC","ABCDEFGHI"
PRINT USING "'LLLLLLLLLLLLLLLL','YOU ONLY SEE PART OF THIS"
PRINT USING "'E","YOU CAN SEE ALL OF THE LINE WHEN IT IS EXTENDED"
END
```

Output

```
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 IT IS EXTENDED
```

14.5 PRINT USING Statement Error Conditions

There are two types of PRINT USING error conditions: fatal and warning. 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 and continues execution.

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. These characters might form another valid field or be considered text. If the invalid characters form a new valid field, a fatal error condition might arise if the item to be printed does not match the field.

The following examples demonstrate invalid character combinations in numeric fields:

Example 1

```
PRINT USING "$$**##.##",5.41,16.30
```

The dollar signs form a complete field and the rest 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".

Output 1

```
$5**16.30
```

Example 2

```
PRINT USING "##.#^^^",5.43E09
```

Because the field has only three carets instead of four, BASIC prints a percent sign and the number, followed by three carets.

Output 2

```
% .543E+10^^^
```

Example 3

```
PRINT USING "'LLEEE","VWXYZ"
```

You cannot combine two letters in one field. BASIC interprets EEE as a string literal.

Output 3

```
VWXEEE
```

15
Handling Run-Time Errors

The process of detecting and correcting errors that occur when your program is running is called **error handling**. This chapter describes default error handling and how to handle HP BASIC run-time errors with your own error handlers.

Throughout this chapter, the term “error” is used to imply any OpenVMS exception, not only an exception of ERROR severity.

15.1 Default Error Handling

HP BASIC provides default run-time error handling for all programs. If you do not provide your own error handlers, the default error handling procedures remain in effect throughout program execution time.

When an error occurs in your program, HP BASIC diagnoses the error and displays a message telling you the nature and severity of the error. There are four severity levels of HP BASIC errors: SEVERE, ERROR, WARNING, and INFORMATIONAL. The severity of an error determines whether or not the program aborts if the error occurs when default error handling is in effect. When default error handling is in effect, ERROR and SEVERE errors always terminate program execution, but program execution continues when WARNING and INFORMATIONAL errors occur.

To override the default error handling procedures, you can provide your own error handlers, as described in the following sections. (Note that you should not call LIB$ESTABLISH from a HP BASIC program as this RTL routine overrides the default error handling procedures and might adversely affect program behavior.)

Only one error can be handled at a time. If an error has occurred but has not yet been handled completely, that error is said to be **pending**. When an error is pending and a second error occurs, program execution always terminates immediately. Therefore, one of the most important functions of an error handler is to *clear* the error so that subsequent errors can also be handled.

If you do not supply your own error handler, program control passes to the HP BASIC error handler when an error occurs. For example, when HP BASIC default error handling is in effect, a program will abort when division by zero is attempted because division by zero is an error of SEVERE severity. With an error handler, you can include an alternative set of instructions for the program to follow; if the zero was input at a terminal, a user-written error handler could display a "Try again" message and execute the program lines again requesting input.

15.2 User-Supplied Error Handlers

It is good programming practice to anticipate certain errors and provide your own error handlers for them. User-written error handlers allow you to handle errors for a specified block of program statements as well as complete program units. Any program module can contain one or more error handlers. These error handlers test the error condition and include statements to be executed if an error occurs.

To provide your own error handlers, you use WHEN ERROR constructs. A WHEN ERROR construct consists of two blocks of code: a protected region and a handler. A **protected region** is a block of code that is monitored by the compiler for the occurrence of an error. A **handler** is the block of code that receives program control when an error occurs during the execution of the statements in the protected region.

There are two forms of WHEN ERROR constructs; in both cases the protected region begins immediately after a WHEN ERROR statement. The following partial programs illustrate each form. In Example 1, the handler is attached to the protected region, while in Example 2, the handler *catch_handler* is detached and must be provided elsewhere in the program unit.

Example 1

```
WHEN ERROR IN
     protected_statement_1
     protected_statement_2
     .
     .
     .
 USE
     handler_statement_1
     handler_statement_2
     .
     .
     .
END WHEN
```

Example 2

```
WHEN ERROR USE catch_handler
     protected_statement_1
     protected_statement_2
     .
     .
     .
END WHEN

HANDLER catch_handler
     handler_statement_1
     handler_statement_2
     .
     .
     .
END HANDLER
```

The following sections further explain the concepts of protected regions and handlers.

15.2.1 Protected Regions

A protected region is a block of code that is monitored by the compiler for the occurrence of an error. The bounds of this region are determined by the actual ordering of the source code. Statements that are lexically between a WHEN ERROR statement and a USE or END WHEN statement are in the protected region.

If an error occurs inside the protected region, control passes to the error handler associated with the WHEN ERROR statement. When an error occurs beyond the limits of a protected region, default error handling is in effect unless other error handlers are provided. For more details about handler priorities, see Section 15.2.3 and Section 15.3.

The WHEN ERROR statement signals the start of a block of protected statements. The WHEN ERROR statement also specifies the handler to be used for any errors that occur inside the protected region. The keyword USE either explicitly names the associated handler for the protected region, or marks the start of the actual handler statements. The statements in the actual error handler receive control only if an error occurs in the protected region.

The following example prompts the user for two integer values and displays their sum. The WHEN ERROR block traps any invalid input values, displays a message telling the user that the input was invalid, and reprompts the user for input.

```
DECLARE INTEGER value_1, value_2

WHEN ERROR IN
     INPUT "PLEASE INPUT 2 INTEGERS"; value_1, value_2 !protected statement
USE
     PRINT "INVALID INPUT - PLEASE TRY AGAIN" !handler statement
     RETRY                                    !handler statement
END WHEN
PRINT "THEIR SUM IS"; value_1 + value_2
```

Protected regions can be nested; a protected region can be within the bounds of another protected region. However, WHEN ERROR statements cannot appear inside an error handler, and protected regions cannot cross over into other block structures. If you are using a WHEN ERROR block with a detached handler, that handler cannot exist within a protected region.

15.2.2 Handlers

A handler is the block of code containing instructions to be executed only when an error occurs during the execution of statements in the protected region. When an error occurs during the execution of a protected region, HP BASIC branches to the handler you have supplied. In turn, the handler processes the error. An error handler typically performs the following functions:

- Determines which error occurred
- Takes appropriate action based on the nature of the error
- Clears the error condition with a RETRY, CONTINUE, END WHEN, or END HANDLER statement
- Continues program execution when possible
- Possibly identifies which program unit or statement caused the error
- Resignals errors with EXIT HANDLER (when an error cannot be handled for some reason)

Handlers can be attached to, or detached from, the statements in the WHEN ERROR protected region.

An **attached handler** is delimited by a USE and an END WHEN statement. The attached handler immediately follows the protected region of a WHEN ERROR IN block. The following example shows an attached handler that traps errors on I/O statements, division by zero, and illegal numbers:

```
PROGRAM accident_prone
 DECLARE REAL age, accidents, rating
 WHEN ERROR IN
 Get_age:
      INPUT "Enter your age";age
      INPUT "How many serious accidents have you had";accidents
      rating = accidents/age
      PRINT "That's ";rating;" serious accidents per year!"
 USE
      SELECT ERR
         !Trap division by zero
         CASE = 61
                  PRINT "Please enter an age greater than 0"
                  CONTINUE Get_age
         !Trap illegal number
         CASE = 52
                  PRINT "Please enter a positive number"
                  RETRY
         CASE ELSE
              !Revert to default error handling
              EXIT HANDLER
      END SELECT
 END WHEN
END PROGRAM
```

A **detached handler** is defined separately in your program unit. It requires an identifier and must be delimited by a HANDLER and an END HANDLER statement. Handler names must be valid HP BASIC identifiers and cannot be the same as the identifier for any label, PROGRAM name, DEF or DEF* function, SUB, FUNCTION, or PICTURE subprogram. The main advantage of using a detached handler is that it can be referenced by more than one WHEN ERROR USE statement. The following example shows a simple detached handler:

```
 WHEN ERROR USE catcher
   KILL "INPUT.DAT"
 END WHEN
    .
    .
    .
 HANDLER catcher
   !Catch if file does not exist
   IF ERR = 5
      THEN CONTINUE
   END IF
 END HANDLER
```

The statements within a handler are never executed if an error does not occur or if no protected region exists for the statement that caused the exception.

When your program generates an error, control transfers to the specified handler. If the code in an error handler generates a second error, control returns to the default HP BASIC error handler and program execution ends, usually with the first error only partly processed. To avoid the possibility of your error handler causing a second error, you should keep handlers as simple as possible and keep operations that might cause errors outside the handler.

Your handler can include conditional expressions to test the error and branch accordingly, as shown in the following example:

```
PROGRAM Check_records
WHEN ERROR USE Global_handler
   .
   .
   .
END WHEN
HANDLER Global_handler
 SELECT ERR
        !Trap buffer overflow
        CASE = 161
            PRINT "Record too long"
            CONTINUE
        !Trap end of file on device
        CASE = 11
            PRINT "End of file"
            CONTINUE
        CASE ELSE
            EXIT HANDLER
 END SELECT
END HANDLER
CLOSE #1%
END PROGRAM
```

Note that ON ERROR statements are not allowed within protected regions or handlers. For compatibility issues related to ON ERROR statements, see Section 15.3.

15.2.3 Exiting from Handlers

After processing an error, a handler typically clears the error so that program execution can continue. HP BASIC provides the following statements that clear the error condition and exit from the handler:

RETRY
CONTINUE
END HANDLER
END WHEN

These statements differ from each other in that they revert control of program execution to different points in the program. Examples of these statements are included in the following sections.

An additional statement, EXIT HANDLER, is provided to allow you to exit from a handler with the error still pending.

The END HANDLER statement identifies the end of the block of statements in the handler. The END WHEN statement marks the end of the protected region when a detached handler is used; it marks the end of the handler when an attached handler is used. If the handler does not process an error with an EXIT HANDLER, RETRY, or CONTINUE statement, the error is cleared by the END HANDLER or END WHEN statement; however, processing continues with the statement immediately after the protected region (and the attached handler, if one exists) where the error occurred. These statements do not return control to the protected region. This is known as "falling out of the bottom of a handler." Be careful not to fall out of the bottom of a handler unintentionally.

Note that you cannot exit from a handler with the following statements:

- EXIT PROGRAM
- EXIT FUNCTION
- EXIT SUB
- EXIT DEF
- GOSUB (with a target outside the handler)
- GOTO (with a target outside the handler)

Also, you cannot exit from a handler with a RESUME statement. The RESUME statement is valid only in blocks of code referred to by ON ERROR statements. Section 15.3 describes the ON ERROR statements.

15.2.3.1 RETRY Statement

You use the RETRY statement to clear the error and to execute the statement again that caused the error again. Be sure to take corrective action before trying the protected statement again. For example:

```
DECLARE REAL radius

WHEN ERROR USE fix_it
     INPUT "Please supply the radius of the circle"; radius
END WHEN
HANDLER fix_it
     !trap overflow error
     IF ERR = 48
     PRINT "Please supply a smaller radius"
     RETRY
END HANDLER
PRINT "The circumference of the circle is "; 2*PI*radius
```

In FOR...NEXT loops, if the error occurs while HP BASIC is evaluating the limit or increment values, RETRY reexecutes the FOR statement; if the error occurs while HP BASIC is evaluating the index variable, RETRY reexecutes the NEXT statement. In UNTIL...NEXT and WHILE...NEXT loops, if the error occurs while HP BASIC is evaluating the relational expression, RETRY reexecutes the NEXT statement.

15.2.3.2 CONTINUE Statement

You can use the CONTINUE statement to clear the error and cause execution to continue at the statement immediately following the propagated error.

When the CONTINUE statement is within an attached handler, you can specify a target. The target can be a line number or label within the bounds of the associated protected region, in a surrounding protected region, or within an unprotected region; however, you must specify a target within the current program module. You cannot specify a target for the CONTINUE statement when it is in a detached handler. For example:

```
DIM LONG her_attributes(10),his_attributes(10)
DECLARE INTEGER counter
WHEN ERROR USE fix_it
  DATA 12,2,35,21,25.5,32,32,30,15,4
  FOR counter = 0 TO 12
      READ her_attributes(counter)
  NEXT counter
  MAT his_attributes = her_attributes
END WHEN
   .
   .
   .
HANDLER fix_it
    !Trap out of data
    IF ERR = 57
       THEN RESTORE
            CONTINUE
    ELSE EXIT HANDLER
    END IF
END HANDLER
```

When a DEF function is invoked from a protected region and an error occurs that has not been handled, a CONTINUE statement with no target causes execution to resume at the statement following the one that invoked the function.

Note that if an error occurs in a loop control statement or SELECT or CASE statement, the CONTINUE statement causes HP BASIC to resume execution at the statement following the end of the loop structure (the NEXT, END CASE, or END SELECT statements).

Note

When you use the RETRY or the CONTINUE statement without a target, the compiler builds read only tables in the generated object file with information about statements in the associated protected regions. Therefore, when space is extremely critical, do not protect large regions with handlers containing RETRY or CONTINUE without a specified target.

15.2.3.3 EXIT HANDLER Statement

Unlike RETRY and CONTINUE, the EXIT HANDLER statement does not clear the error; rather, it allows you to exit from the handler with the error pending. This allows you to pass an error to the handler associated with the next outer protected region, or back to HP BASIC default error handling, or to the calling procedure.

When an error occurs within a nested protected region, control passes to the handler associated with the innermost protected region in which the error occurred. If the innermost handler does not handle the error, the error is passed to the next outer handler with the EXIT HANDLER statement. All handlers for any outer WHEN ERROR blocks are processed before reverting to default error handling or resignaling the calling procedure.

The following example shows two nested protected regions. Neither handler traps division by zero. If division by zero occurs, the handler associated with the innermost protected region, *inner_handler*, does not clear the error; therefore, the error is passed to the handler associated with the next outer protected region. *Outer_handler* does not clear this error either, and so the error is passed to the default error handler. This error is fatal and the program ends abnormally. Output is specific to VAX BASIC.

```
PROGRAM nesting
 OPTION TYPE = EXPLICIT
 DECLARE LONG divisor
 DECLARE REAL dividend, quotient
 WHEN ERROR USE outer_handler
      INPUT "Enter divisor";Divisor
      INPUT "Enter dividend";Dividend

      WHEN ERROR USE inner_handler
           Quotient = Dividend/Divisor
           PRINT "The quotient is ";Quotient
      END WHEN

 END WHEN
 HANDLER outer_handler
         !Trap data format error
         IF ERR = 50
            THEN
            PRINT "Illegal input...try again"
            RETRY
            ELSE PRINT "In outer_handler"
                 PRINT "Reverting to default handling now"
                 EXIT HANDLER
         END IF
 END HANDLER
```

```
 HANDLER inner_handler
         !Trap overflow/decimal error
         IF ERR = 181
            THEN CONTINUE
            ELSE PRINT "Inside inner_handler"
                 PRINT "Reverting to outer handler now"
                 EXIT HANDLER
         END IF
 END HANDLER
END PROGRAM
```

For more information about exiting program units while an error is pending, see Section 15.2.6.

15.2.4 Selecting the Severity of Errors to Handle

The OPTION HANDLE statement lets you specify the severity level of errors that are to be handled by an error handler in addition to the BASIC errors that can normally be handled or trapped. You can specify any one of the following error severity levels: BASIC, SEVERE, ERROR, WARNING, or INFORMATIONAL.

OPTION HANDLE = BASIC is the default, which is in effect if you do not specify an alternative in the OPTION HANDLE statement. Only HP BASIC errors that can be trapped transfer control to the current error handler when this option is in effect. Refer to Appendix B to determine which BASIC errors cannot be trapped.

When you specify an error severity level other than BASIC in the OPTION HANDLE statement, the following errors will transfer control to the error handler:

- All BASIC errors that can be trapped of this or lesser severity
- All non-BASIC errors of this or lesser severity
- BASIC errors of this or lesser severity that normally cannot be trapped

For example, if you specify OPTION HANDLE = ERROR, you can handle all BASIC and non-BASIC errors of ERROR severity (both those that can and those that cannot be trapped), and all WARNING and INFORMATIONAL errors, but no SEVERE errors.

15.2.5 Identifying Errors

HP BASIC provides several built-in functions that return information about an error. You can use these functions inside your error handlers to determine details about the error and conditionally handle these errors. These functions include:

ERR
ERL
ERN$
ERT$
VMSSTATUS
RMSSTATUS

Note that if an error occurs in your program that is not a HP BASIC error or does not map onto a HP BASIC error, it is signaled as NOTBASIC ("Not a BASIC error" (ERR=194). In this case, you can use the built-in function VMSSTATUS to determine what caused the error.

15.2.5.1 Determining the Error Number (ERR)

You use the ERR function to return the number of the last error that occurred. Appendix B lists the number of each HP BASIC run-time error—for example, ERR 153 is "RECALREXI, Record already exists."

```
OPTION HANDLE = ERROR
WHEN ERROR USE find_error
   .
   .
   .
END WHEN

HANDLER find_error
  SELECT ERR
        !Record already exists
        CASE = 153
                PRINT "Choose new record"
                CONTINUE
        CASE ELSE
                EXIT HANDLER
  END SELECT

END HANDLER
```

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.

15.2.5.2 Determining the Error Line Number (ERL)

After your program generates an error, the ERL function returns the BASIC line number of the signaled error. This function is valid only in line-numbered programs. The ERL function, like ERR, lets you set up branching to one of several paths in the code.

In the following example, the handler continues execution at different points in the program, depending on the value of ERL:

```
10 DECLARE INTEGER CONSTANT TRUE = -1
20 WHEN ERROR USE err_handler
   .
   .
   .
900 END WHEN
1000 HANDLER err_handler
       SELECT TRUE
           CASE (ERR = 11) AND (ERL = 790)
               !Is error end of file at line 790?
               PRINT "Completed"
               CONTINUE
           CASE (ERR = 149) AND (ERL = 80)
               !Is error not at end of file on line 80?
               PRINT "CHECK ACCESS MODE"
               CONTINUE
           CASE ELSE
               !Let BASIC handle any other errors
               EXIT HANDLER
1500   END SELECT
2000 END HANDLER
32000   CLOSE #5
32767   END
```

The results of ERL are undefined until an error occurs, or if the error occurs in a subprogram not written in HP BASIC. Although ERL remains defined as the line number of the last error even after control leaves the error handler, it is poor programming practice to refer to this variable outside the scope of an error handler.

If you reference ERL in a compilation unit with line numbers, code and data are included in your program to allow HP BASIC to determine ERL when an exception occurs. If you do not need to reference ERL, you can save program size and reduce execution time by compiling your program with the /NOLINE qualifier. Alpha BASIC uses the /NOLINE qualifier by default to compile programs. Even if you do not use any line numbers, you can reduce execution time by compiling with the /NOLINE qualifier.

If an error occurs in a subprogram containing line numbers, HP BASIC sets the ERL variable to the subprogram line number where the error was detected. If the subprogram also executes an EXIT HANDLER statement, control passes back to the outer procedure's handler. The error is assumed to occur on the statement where the call or invocation occurs.

15.2.5.3 Determining Where the Error Occurred (ERN$)

You use the ERN$ function to return the name of the program unit in which the error was detected. ERN$ returns the name of a main program, SUB, FUNCTION, or PICTURE subprogram, or DEF function. If the PROGRAM statement is used with a user-supplied identifier, the ERN$ value is the specified identifier for the main program. The results of ERN$ are undefined until the program generates an error.

In the following example, control passes to the main program for error handling if the error occurs in the module SUBARC:

```
HANDLER locat_ern
  IF ERN$ = "SUBARC"
     THEN PRINT "ERROR IS ";ERR
          PRINT "RETURNING TO MAIN PROGRAM FOR ERROR HANDLING"
          EXIT HANDLER
     ELSE  PRINT "PROGRAM MODULE GENERATING ERROR IS ";ERN$
  END IF
END HANDLER
```

Note that ERN$ is invalid when an error occurs in a subprogram compiled with the /NOSETUP qualifier.

15.2.5.4 Determining the Error Message Text (ERT$)

You use the ERT$ function to access the message text associated with a specified error number. Use of the ERT$ function is not limited to the scope of the error handler; you can access ERT$ at any time. The following detached handler tests whether the error occurred in a DEF module named TSLFE, and, if so, prints the text of the signaled error and resumes execution:

```
HANDLER catch_it
  IF ERN$ = "TSLFE"
        THEN PRINT ERT$(ERR)
        CONTINUE
        ELSE EXIT HANDLER
  END IF
END HANDLER
```

15.2.5.5 Determining OpenVMS Error Information

HP BASIC provides a built-in function, VMSSTATUS, that returns the originally signaled error before it is translated to a BASIC error. For example, for the BASIC error "End of file on device" (ERR=11), the VMSSTATUS function returns "RMS$_EOF" (RMS end of file). This function is useful when the error is NOTBASIC (ERR=194).

When there is no error pending, VMSSTATUS is undefined. The value returned by this function is the actual signaled error value. If non-BASIC errors are being handled, the VMSSTATUS function might be the only way to find out which error caused the exception.

The following example shows a program that performs file I/O. The first WHEN ERROR block traps any errors that occur while the program is opening the file or requesting user input. The detached handler for this block checks the value of VMSSTATUS to determine the exception that occurred. The inner error handler handles two special errors, BAS$K_RECNOTFOU and BAS$K_RECBUCLOC, separately. If the error signaled does not correspond to one of these, the inner error handler passes control to the outer handler with the EXIT HANDLER statement. The outer handler sets the program status to VMSSTATUS. When the program exits, the operating system displays any status that is of warning severity or greater.

```
PROGRAM Tester

 OPTION HANDLE = ERROR
 EXTERNAL LONG CONSTANT BAS$K_RECNOTFOU, BAS$K_RECBUCLOC
 DECLARE LONG Final_status
 MAP (Rec_buffer)                          &
      STRING Rec_key        = 5,           &
      STRING Rest_of_record = 20
 Final_status = 1
 WHEN ERROR USE Global_handler
      OPEN "My_database" FOR INPUT AS FILE #1  &
           ,INDEXED FIXED                      &
           ,ACCESS READ                        &
           ,MAP Rec_buffer                     &
           ,PRIMARY Rec_key
 Get_key:
     INPUT "Record to retrieve"; Rec_key
     WHEN ERROR IN
          GET #1%, KEY #0 EQ Rec_key
          PRINT Rest_of_record
     USE
          SELECT ERR
              CASE = BAS$K_RECNOTFOU
                     PRINT "Record not found"
                     CONTINUE Get_key
```

```
            CASE = BAS$K_RECBUCLOC
                   SLEEP 2%
                   RETRY
            CASE ELSE
                 EXIT HANDLER
         END SELECT
     END WHEN
    END WHEN
    HANDLER Global_handler
            Final_status = VMSSTATUS
    END HANDLER

END PROGRAM Final_status
```

15.2.5.6 Determining RMS Error Information

The RMSSTATUS function lets you determine which RMS error caused a resulting HP BASIC error. You must specify an open channel as the first parameter to RMSSTATUS. If this channel is not open, the error "I/O channel not open" (ERR=9) is signaled. The second parameter to the function lets you specify either STATUS or VALUE; this parameter is optional. If you do not specify the second parameter, RMSSTATUS returns the STATUS value by default. STATUS represents the RMS STS field and VALUE corresponds to the RMS STV field.

The following example shows an error handler that prints both the status and the value of any RMS error:

```
WHEN ERROR IN
     OPEN "file.txt" FOR OUTPUT AS FILE 1%
     PRINT #1%, TIME$(0%)
USE
     !Error 12 is fatal system I/O failure
     IF ERR = 12
        THEN
        PRINT "An unexpected RMS error has occurred:"
        PRINT "Status = "; RMSSTATUS(1%)
        PRINT "Value = "; RMSSTATUS(1%, VALUE)
        EXIT HANDLER
     END IF
END WHEN

CLOSE #1%
GOTO done

done:
    END
```

If you want to find an RMS status without knowing which particular channel to check, you can use VMSSTATUS to get the STATUS value (STS) if an error has occurred.

15.2.6 Ctrl/C Trapping

Error handling procedures are commonly used to trap user Ctrl/C responses. With Ctrl/C trapping enabled, control is transferred to an error handler if a user presses Ctrl/C during program execution. You enable Ctrl/C trapping in your program by invoking the built-in CTRLC function. For example:

```
Y% = CTRLC
```

After you invoke the CTRLC function, a Ctrl/C entered at the terminal transfers control to the error handler. Once the Ctrl/C is trapped, you can include routines to interact with the program, as shown in the following example:

```
WHEN ERROR IN
  Y% = CTRLC
  OPEN 'FIL_DAT' FOR INPUT AS FILE #1%
  INPUT "HOW MANY RECORDS"; Rec_read%
  FOR I% = 1% TO Rec_read%
    GET #1%
    PRINT Name$, Address$, Emp_code%
    PRINT
  NEXT I%

USE
  !Trap ^C
  IF (ERR = 28%)
     THEN PRINT "CURRENT RECORD IS "; I%
     ELSE EXIT HANDLER
  END IF
  CONTINUE Clean_up
END WHEN
   .
   .
   .
Clean_up:
  CLOSE #1%
  PRINT "END OF PROCESSING"
END
```

Output

```
SMITH, DEXTER 231 COLUMBUS ST            09341

TRAVIS, JOHN  PO BOX 80                  64119

^C

THE CURRENT RECORD IS 3

END PROCESSING
```

Note that the error condition is still pending until the error handler executes the CONTINUE statement. Therefore, if you press Ctrl/C a second time while the error handler is executing, control returns to the HP BASIC error handler, which terminates the program.

To disable Ctrl/C trapping, use the RCTRLC function. The RCTRLC function disables only Ctrl/C trapping, not the Ctrl/C interrupts themselves.

15.2.7 Handling Errors in Multiple-Unit Programs

You can use WHEN ERROR constructs anywhere in your main program or program modules. Procedure and function invocations, such as invocations of DEF and DEF* functions and SUB, FUNCTION, and PICTURE subroutines, as well as non-BASIC programs, are valid within protected regions. GOTO and GOSUB statements are valid within handlers provided that the target is within the handler, an outer handler, or an unprotected region. Note, however, that a detached handler cannot appear within DEF or DEF* functions without the associated protected region.

When an error occurs within nested protected regions, HP BASIC maintains the same priorities for handler use; control always passes to the handler associated with the innermost protected region in which the error occurred. When an exception occurs, all handlers for any outer WHEN ERROR blocks are processed before the program reverts to default error handling. Outer handlers are invoked when an inner handler executes an EXIT HANDLER statement. When there are no more outer handlers, and the outermost handler executes an EXIT HANDLER statement, program control reverts to the handler associated with the calling routine. For example:

```
SUB LIST(A$)
  WHEN ERROR USE sub_handler

      OPEN A$ FOR INPUT AS FILE #12%
  Get_data:
      LINPUT #12%, B$
      PRINT B$
      GOTO Get_data
  END WHEN
  HANDLER sub_handler
    !Trap end of file
    IF ERR <> 11%
        THEN EXIT HANDLER
    END IF
END HANDLER
CLose_up:
   CLOSE #12%
END SUB
```

You can call a subprogram while an error is pending; however, if you do, the subprogram cannot resignal an error back to the calling program. If the subprogram tries to resignal an error, HP BASIC signals “Improper error handling” and program execution terminates.

The following rules apply to error handling in function definitions:

- DEF and DEF* function definitions cannot appear within a protected region. However, protected regions can be contained within the function definitions.
- To trap errors while a DEF function is active, include protected regions inside the DEF function. If you do this, the associated handler remains in effect until your program leaves the protected region, or the DEF function.

For example:

```
WHEN ERROR IN
.
.
.
Invoke_def:
  A% = FNIN_PUT%("PROMPT")

USE
   PRINT "ERROR"; ERT$(ERR%);
   IF ERN$ = "FNIN_PUT"
      THEN PRINT "IN FUNCTION"
           CONTINUE
      ELSE PRINT "IN MAIN"
           CONTINUE Invoke_def
   END IF
END WHEN

Main_code:
DEF FNIN_PUT%(P$)
 WHEN ERROR IN
   PRINT P$
   INPUT LINE_IN$
   FNIN_PUT% = INTEGER(LINE_IN$)
USE
    IF ERR = 50
       THEN RETRY
       ELSE EXIT HANDLER
    END IF
END WHEN
END DEF
```

Note

If you invoke a GOSUB statement or a DEF* function from within a protected region and the invoked procedure is outside of any protected region, all pending errors are handled by the WHEN ERROR handler unless a previously executed ON ERROR statement specifies otherwise.

15.2.8 Forcing Errors

The CAUSE ERROR statement allows a program to artificially generate an error when the program would not otherwise do so. You can force any HP BASIC run-time error. You must specify the number of the error the compiler should force; the error numbers are listed in Appendix B. The following statement forces an end-of-file error (ERR=11) to occur:

```
CAUSE ERROR 11%
```

You can use this feature to debug an error handler during program development, as shown in the following example:

```
WHEN ERROR IN
   .
   .
   .
 CAUSE ERROR 11%
   .
   .
   .
USE
   SELECT ERR
        CASE = 11%
             PRINT "Trapped an end of file on device"
             CONTINUE
        CASE ELSE
             EXIT HANDLER
END WHEN
```

15.3 Using the ON ERROR Statements

HP BASIC supports ON ERROR statements as an alternative to WHEN blocks primarily for compatibility with existing programs. WHEN ERROR blocks are similar to declarative statements in that they do not depend on run-time flow of control. The ON ERROR statements, however, affect error handling only if the statements execute at run time. For example, if a GOTO statement precedes an ON ERROR statement, the ON ERROR statement will not have any effect because it does not execute.

WHEN ERROR blocks let you handle errors that occur in a specific range of statements. ON ERROR statements let you specify a general error handler that is in effect until you specify another ON ERROR statement or until you pass control to the HP BASIC error handler.

Note

For all current program development, it is recommended that you use WHEN ERROR constructs for user-written error handlers. Mixing WHEN ERROR constructs and ON ERROR statements within the same program is not recommended. The ON ERROR statements are supported for compatibility with other versions of BASIC available from HP. It is important to note that all of these statements are illegal within a protected region, or an attached or detached handler.

The ON ERROR statements are documented in the *HP BASIC for OpenVMS Reference Manual*. This section briefly describes the main features of the ON ERROR statements.

The ON ERROR statements can be used to transfer control to a labeled block of error handling code. If you have executed an ON ERROR statement and an error occurs, the ON ERROR statement immediately transfers control to the label or line number that starts the error handling code. Otherwise, the ON ERROR statement specifies the branch to be taken in the event of an error.

There are three forms of the ON ERROR statement:

- ON ERROR GOTO 0

  The ON ERROR GOTO 0 statement reverts control to HP BASIC default error handling in one of two ways:

  - If an error is pending, execution of the ON ERROR GOTO 0 statement returns control to the HP BASIC error handler immediately.
  - If no error is pending, an ON ERROR GOTO 0 statement disables your current error handler. The HP BASIC error handler handles all subsequent errors until another ON ERROR statement is executed, unless an error occurs in a WHEN ERROR protected region.

- ON ERROR GOTO *target*

  The ON ERROR GOTO *target* statement reverts control to the target when subsequent errors occur that are not handled by WHEN block handlers.

- ON ERROR GO BACK

  The ON ERROR GO BACK statement transfers control to the calling program's error handler if an error occurs in the subprogram or DEF function. If you use ON ERROR GO BACK in a PROGRAM unit (outside of a DEF function) and no other outer protected region exists, it is equivalent to ON ERROR GOTO 0 and HP BASIC default error handling is in effect. With ON ERROR GO BACK, if an error occurs in the execution of a function or subprogram, the error is passed to either the error handler of the surrounding program module (in the case of a DEF function definition) or to the error handler of the calling program (in the case of a separately compiled subprogram).

  An error handler in the DEF function does not permanently override an error handler in the main program. HP BASIC saves the error handler in the main program when you transfer into a DEF, and restores it when you return.

The ON ERROR GOTO statement is usually placed before any other executable statements. The following example clears end-of-file errors and passes all other errors back to the HP BASIC default error handling procedures:

```
5 ON ERROR GOTO Error_handler
   .
   .
   .
Error_handler:
    !Trap end of file on device
    IF ERR = 11
        THEN
                RESUME 1000
        ELSE
                ON ERROR GO BACK
    END IF
```

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 error handler.

Every ON ERROR error handler must end with one of the following statements:

- RESUME [*target*]
- ON ERROR GOTO 0
- ON ERROR GO BACK

If none of these statements is present, the HP BASIC error handler aborts your program with the fatal error "Error trap needs RESUME" as soon as an END, END SUB, END DEF, END FUNCTION, END PROGRAM, or END PICTURE statement is encountered. The RESUME statement, like the RETRY and CONTINUE statements, clears the error condition.

You can resume execution at any line number or label that is in the same module as the RESUME statement, unless that line or target is inside a DEF function, a WHEN ERROR protected region, or a handler. In general, RESUME without a target transfers control to the beginning of the program block where the error occurred.

- If you resume execution at a multistatement line, execution begins at the first statement after the line number or label—not necessarily at the statement that generated the error.
- If an entire loop block is associated with a single line number or label and an error occurs within the loop, RESUME with no target transfers control to the statement immediately *after* the FOR, WHILE, or UNTIL statement, not to the line number or label.

For more information about the RESUME statement, see the *HP BASIC for OpenVMS Reference Manual*.

Using both ON ERROR statements and WHEN ERROR constructs in the same program is not recommended. However, when this is the case, the order of handler priorities is as follows:

1. Control passes to the handler associated with the innermost WHEN ERROR block.
2. If protected regions are nested, the pending error is handled by the handler associated with the next outer WHEN ERROR block.
3. When no outer protected regions can handle the error, and if an ON ERROR statement is in effect, control transfers to the target of the next outer ON ERROR statement (if one is present).
4. If no outer handler is available or can handle the error, the error is passed to HP BASIC default error handling. Default error handling is equivalent to ON ERROR GOTO 0 for main procedures, and ON ERROR GO BACK for SUBs, FUNCTIONs, and DEFs.

For information about specific run-time errors, see Appendix B.

16
Compiler Directives

Compiler directives are instructions that tell HP BASIC to perform certain operations as it translates a source program. This chapter describes how to control program compilation using compiler directives.

16.1 Overview of Compiler Directives

With compiler directives, you can do the following:

- 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 the 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 HP BASIC code from another source file or a text library.
- Include CDD/Repository record definitions in a HP BASIC program.
- Record dependency relationships in CDD/Repository.
- Display a message at compile time.
- Conditionally compile parts of a program.
- Terminate compilation.

When using compiler directives, follow these rules:

- Directives must begin with a percent sign (%).
- Directives can be preceded by an optional line number.
- Directives must be the only text on the line (except for %IF-%THEN-%ELSE-%END %IF).

- Directives cannot appear within a quoted string.
- Directives cannot follow an END, END SUB, or END FUNCTION statement.

16.2 Controlling the Compilation Listing

The following compiler listing directives let you control the content and appearance of the compilation listing:

- %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.

These directives are described in the following sections.

The 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.

16.2.1 %TITLE and %SBTTL Directives

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 is a quoted string of up to 31 characters and normally contains the source program title and other information.

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.

%TITLE must appear on its own line. For example:

```
%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. If you want the subtitle to appear on the first page, the %SBTTL directive must appear directly after the %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. As in %TITLE, the text you use in %SBTTL must be a quoted string not exceeding 31 characters. Note, however, that subtitle information appears on listing pages that contain the actual source code.

The following example shows the use of both %TITLE and %SBTTL directives. 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, the second line on each subsequent page becomes "Subroutines." When BASIC encounters the %SBTTL directive, the second line on each subsequent page becomes "Error Handler."

```
%TITLE "Payroll Program"
%SBTTL "Constant Declarations"
   .
   .
   .
%SBTTL "Subroutines"
   .
   .
   .
%SBTTL "Error Handler"
   .
   .
   .
```

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.

16.2.2 %IDENT Directive

The %IDENT directive identifies the version of a program module. The identification text must be a quoted string of up to 31 characters. The information contained within the identification text appears in the listing file and the object module. Thus, the map file created by the OpenVMS Linker also contains this information.

The identification text appears in the first 31 character positions of the second line on each subsequent listing page. In the following example, 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 linker generates a map listing, this information also appears there.

```
%IDENT "V5.3"
SUB PAY
   .
   .
   .
```

If your source module contains multiple %IDENT directives, BASIC signals a warning and uses the version specified in the first %IDENT directive.

16.2.3 %PAGE Directive

The %PAGE directive causes BASIC to begin a new page in the listing file. In the following example, the %PAGE directives cause BASIC to skip to a new page in the listing file just before each new subtitle. Note that, to have title and subtitle information appear in the heading of each page, you cannot place a line number between the %PAGE, %TITLE, and %SBTTL directives.

```
%TITLE "Payroll Program"
%SBTTL "Constant Declarations"
   .
   .
   .
%PAGE
%SBTTL "Subroutines"
   .
   .
   .
%PAGE
%SBTTL "Error Handler"
   .
   .
   .
```

16.2.4 %LIST and %NOLIST Directives

%LIST and %NOLIST are complementary directives. The %LIST directive causes BASIC to resume adding information to the listing file, while the %NOLIST directive causes BASIC to stop adding information to the listing file. Therefore, you can control which parts of the source program are to be listed.

In the following example, when BASIC encounters the %LIST directive, it resumes adding new information to the listing file:

```
%TITLE "Payroll Program"
%SBTTL "Constant Declarations"
   .
   .
   .
%NOLIST
   .
   .
   .
%LIST
   .
   .
   .
%PAGE
%SBTTL "Subroutines"
   .
   .
   .
%PAGE
%SBTTL "Error Handler"
   .
   .
   .
```

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.

16.2.5 %CROSS and %NOCROSS Directives

The %CROSS and %NOCROSS directives are complementary. The %CROSS directive causes BASIC to resume adding cross-reference information, while the %NOCROSS directive causes BASIC to stop adding cross-reference information to the listing file. Therefore, you can specify that only certain parts of the source program are to be cross-referenced.

In the following example, as soon as BASIC encounters the %CROSS directive, it resumes adding new cross-reference information to the listing file:

```
%TITLE "Payroll Program"
%SBTTL "Constant Declarations"
   .
   .
   .
%NOCROSS
   .
   .
   .
%CROSS
   .
   .
   .
%PAGE
%SBTTL "Subroutines"
   .
   .
   .
%PAGE
%SBTTL "Error Handler"
   .
   .
   .
```

If you have not requested the creation of a cross-reference listing, the %CROSS and %NOCROSS directives have no effect.

16.3 Accessing External Source Files

The %INCLUDE directive lets you access BASIC source text from a file into the source program. The %INCLUDE directive also lets you access record definitions in CDD/Repository as well as access source text from a text library. The line on which a %INCLUDE directive resides can be continued, but cannot contain any other directives or statements.

If you are including a source text file, you must supply a file specification. If you do not provide a file type, BASIC uses the default type .BAS. For example:

```
%INCLUDE  "KEN.BAS"
```

If you are including a CDD/Repository definition, you must supply a valid CDD/Plus path specification to extract a RECORD definition from CDD/Repository. For example:

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

See the *CDD/Repository CDO Reference Manual* for more information about CDD/Repository.

If you are including source text from a text library, you must supply the name of the text module you wish to include as well as the name of the library where the module resides. If you do not specify a library name, BASIC uses the default library, BASIC$LIBRARY. Moreover, if you do not specify a directory name or file type, BASIC uses the default device and the file type .TLB. If the BASIC$LIBRARY logical name is undefined, SYS$LIBRARY:BASIC$STARLET.TLB is used. The default file specification is BASIC.TLB.

In the following example, when BASIC encounters the %INCLUDE directive, the compiler searches through the library SYS$LIBRARY:BASIC_LIB.TLB for the specified module DMB_TEST and compiles the text as if it were placed in the position of the %INCLUDE directive:

```
%INCLUDE "DMB_TEST" %FROM %LIBRARY "SYS$LIBRARY:BASIC_LIB.TLB"
```

BASIC supplies the text library BASIC$STARLET located in SYS$LIBRARY. This text library contains condition codes and other symbols defined in the system object and shareable image libraries. Using the definitions from BASIC$STARLET allows you to reference condition codes and other system-defined symbols as local, rather than global symbols.

To create your own text libraries using the OpenVMS Librarian utility, see the *VMS Librarian Utility Manual*.

All file specifications, CDD/Repository path specifications, text modules, and library specifications must be string literals enclosed in quotation marks.

The source files accessed with %INCLUDE cannot contain line numbers. This requirement means that all statements in the accessed file are associated with the BASIC line containing the %INCLUDE directive if line numbers are being used. Therefore, if you are using line numbers, 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.

When a program is compiled, BASIC inserts the included text at the point at which it encounters 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. "I*n*" specifies text that was either accessed from a source file or from a text library, and "C*n*" specifies a record definition that was accessed from CDD/Repository. Both the I and the C tell you that the text was accessed with the %INCLUDE directive, and *n* tells you the nesting level of the included text.

The %INCLUDE directive is useful when you want to share code among multiple program modules. To do this, you must first create a file that contains the shareable code, then include that file in all the modules that require it. Thus, you reduce the chance of a typographical error.

You can prevent the %INCLUDE file code from appearing in the compilation listing by using the BASIC command qualifier /SHOW=NOINCLUDE or /SHOW=NOCDD_DEFINITIONS. For text files and text library modules, use the qualifier /SHOW=NOINCLUDE. For CDD/Repository definitions, use the qualifier /SHOW=NOCDD_DEFINITIONS.

16.4 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 LONG integers.

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.
- Display informational messages during the compilation.

BASIC also supplies the lexical built-in function %VARIANT that can be used to conditionally control compilation.

%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 directive)
- %VARIANT
- %ABORT
- %PRINT
- %IF-%THEN-%ELSE-%END %IF

16.4.1 %LET Directive

The %LET directive creates and assigns values to lexical constants. Lexical constants are always LONG integers. These constants control the execution of the %IF-%THEN-%ELSE-%END %IF directive.

All lexical constants must be created with %LET before they can be used in a %IF-%THEN-%ELSE-%END %IF directive, and each lexical constant must be created with a separate %LET directive. All lexical constant names must also be preceded by a percent sign and cannot end with a dollar sign or percent sign.

A lexical expression can be any of the following:

- 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:

```
%LET %debug_on = 0%
```

See Section 16.4.5 for an example of using %LET with %IF-%THEN-%ELSE.

16.4.2 %VARIANT Directive

The %VARIANT directive is a built-in lexical function that returns an integer. The value of this returned integer is determined by:

- The SET VARIANT command when a program is compiled in the VAX BASIC Environment
- The /VARIANT qualifier when a program is compiled from the system command level or from within the VAX BASIC Environment

The %VARIANT function returns the variant value set with either of these methods.

The default value for the %VARIANT function is zero. See Section 16.4.5 for an example of controlling compilation with %VARIANT.

16.4.3 %ABORT Directive

The %ABORT directive terminates the compilation and displays a message you provide.

The text must be a quoted string literal. This information is displayed to SYS$ERROR and in the compilation listing if one is being created. BASIC stops the compilation and terminates the listing file as soon as it encounters a %ABORT directive, and so BASIC does not perform syntax checking on the remainder of the program. See Section 16.4.5 for an example of using %ABORT.

16.4.4 %PRINT Directive

The %PRINT directive allows you to insert a message into your source code that the BASIC compiler displays at compile time.

The text must be a quoted string literal. This information is displayed to SYS$ERROR and in the compilation listing if one is being created. BASIC prints the message specified as soon as it encounters a %PRINT directive. See Section 16.4.5 for an example of using %PRINT.

16.4.5 %IF-%THEN-%ELSE-%END %IF Directive

The %IF-%THEN-%ELSE-%END %IF directive lets you do the following things conditionally:

- Compile source text
- Execute another compiler directive

This directive differs from all others in that it can appear anywhere in a program where a space is allowed, except within a quoted string.

You must include %END %IF. Otherwise, the rest of the source program becomes part of the %THEN or %ELSE clause. You must also include a lexical expression and some BASIC source code.

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 does not compile the source code in the %ELSE clause. If the lexical expression is false, BASIC does not compile the source code in the %THEN clause. However, HP BASIC does check for lexical errors (such as illegally formed numeric constants) in the uncompiled block of code. If an uncompiled block of code contains a lexical error, HP BASIC signals an error.

Even though HP BASIC compiles only one block of code in an %IF-%THEN-%ELSE-%END-%IF directive, you cannot use the same line number in both a %THEN block and an %ELSE block. If you specify the same line number, the first occurrence of the line number is replaced by the second when the program is compiled.

The following example uses the %VARIANT directive, which returns the value set by the SET VARIANT command or /VARIANT qualifier:

```
%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:

```
DECLARE %IF (%VARIANT=2%) %THEN LONG %ELSE WORD %END %IF int_array(100)
```

A %THEN or %ELSE clause can also contain other compiler directives. For example, the following program creates the lexical constant *%my_constant* and assigns it a value of 8. 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 the expression is false, BASIC prints the specified message and continues to compile your program without aborting the compilation.

```
%LET %my_constant = 8%
%IF ( (%my_constant + %VARIANT) >= 10% )%THEN
    %ABORT "Cannot compile with VARIANT >= 2"
    %ELSE
    %PRINT "Successful Compilation"
%END %IF
```

The compilation listing shows you which clause was actually compiled.

16.4.6 %DEFINE and %UNDEFINE Directives

The %DEFINE directive allows you to assign a value to an identifier. The %UNDEFINE directive will remove the value.

The representation of this value stays in force until a corresponding %UNDEFINE directive or the end of the source module is encountered.

16.5 Record Dependency Relationships in CDD/Repository

By using the %INCLUDE %FROM %CDD or the %REPORT %DEPENDENCY directives in conjunction with the /DEPENDENCY_DATA qualifier in the BASIC command, you can record dependency relationships in a CDO dictionary between a compiled module entity and included records or other referenced dictionary entities.

See Chapter 21 for more information about record dependency relationships.

17
Data Representation

This chapter describes how HP BASIC represents data stored in memory.

The following sections discuss four types of data representation: integer, float, decimal, and string.

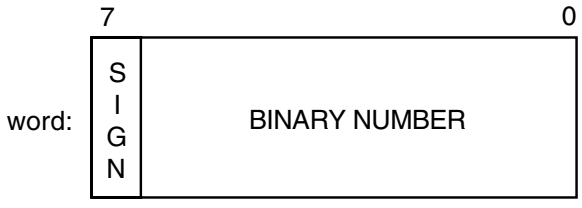
17.1 Integer Format

There are four ways in which integer data can be represented, depending on the size of the data to be stored: byte, word, longword, and quadword. Negative integer values are stored in two's complement format. The following sections describe each of these formats.

17.1.1 Byte-Length Integer Format

Byte-length integers are in the range -128 to 127 and are stored as 1 byte (8 bits), starting on an arbitrary byte boundary. Bits are labeled from the right, 0 to 7, as in Figure 17–1.

Figure 17–1 Byte-Length Integer Format

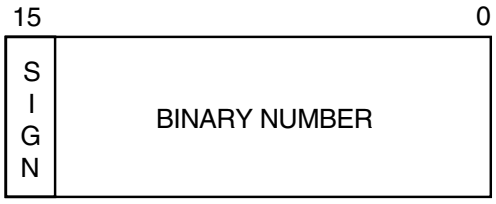


ZK–5173–GE

17.1.2 Word-Length Integer Format

Word-length integers are in the range -32768 to 32767 and are stored as two contiguous bytes, starting on an arbitrary byte boundary. Bits are labeled from the right, 0 to 15, as in Figure 17–2.

Figure 17–2 Word-Length Integer Format

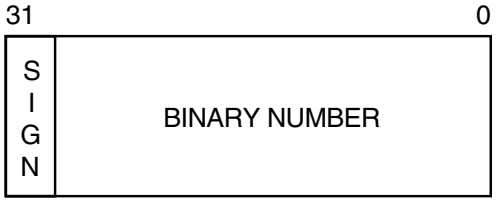


ZK–5174–GE

17.1.3 Longword Integer Format

Longword integers are stored as four contiguous bytes, starting on an arbitrary byte boundary. Values are in the range -2147483648 to 2147483647. See Figure 17–3.

Figure 17–3 Longword Integer Format

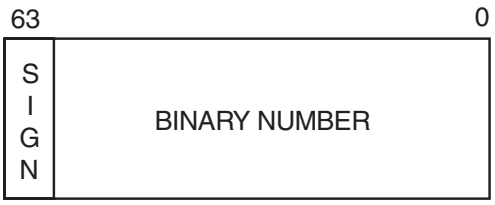


ZK–5175–GE

17.1.4 Quadword Integer Format

Quadword integers are stored as eight contiguous bytes, starting on an arbitrary byte boundary. Values are in the range -9223372036854775808 to 9223372036854775807. See Figure 17–4.

Figure 17–4 Quadword Integer Format



VM-0520A-AI

The compiler incorrectly gives an integer overflow message when the most negative integer constants are used, as follows:

```
BYTE -128%
WORD -32768%
LONG -2147483648%
QUAD -9223372036854775808%
```

The workaround is to use the appropriate expression from the following:

```
BYTE -127% - 1%
WORD -32767% - 1%
LONG -2147483647% - 1%
QUAD -9223372036854775807% - 1%
```

17.2 Real Number Format

Real numbers, like integers, can be represented in varying formats, depending on the size of the data to be stored. These formats include SINGLE floating-point, DOUBLE floating-point, GFLOAT floating-point, SFLOAT floating-point, TFLOAT floating-point, XFLOAT floating-point, and packed DECIMAL format. The following sections describe each of these formats.

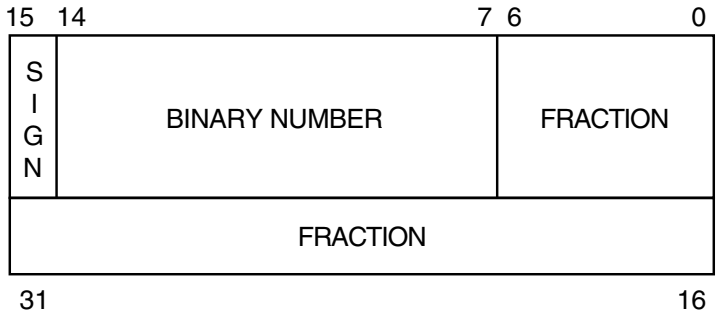
17.2.1 SINGLE Floating-Point Number Format (F_floating)

F_floating (single-precision) floating-point numbers are stored as four contiguous bytes, starting on an arbitrary byte boundary. Bits are labeled from the right, 0 to 31.

The format for single-precision is sign magnitude, with bit 15 the sign bit, bits 14 to 7 an excess-128 binary exponent, and bits 6 to 0 and 31 to 16 a normalized 24-bit fraction with the redundant, most significant fraction bit not represented. See Figure 17–5 for the format. The 8-bit exponent field encodes the values from 0 to 255, inclusively.

An exponent value of 0 together with a sign bit of 0 indicates that the F_floating number has a value of 0. Exponent values from 1 to 255 indicate true binary exponents of -127 to 127. An exponent value of 0, together with a sign bit of 1, is taken as reserved. (Floating-point instructions processing a reserved operand take a reserved operand fault.) The magnitude of an F_floating number is in the approximate range $.29 * 10^{-38}$ to $1.7 * 10^{38}$. The precision of an F_floating number is approximately one part in 2^{23} (approximately 7 decimal digits).

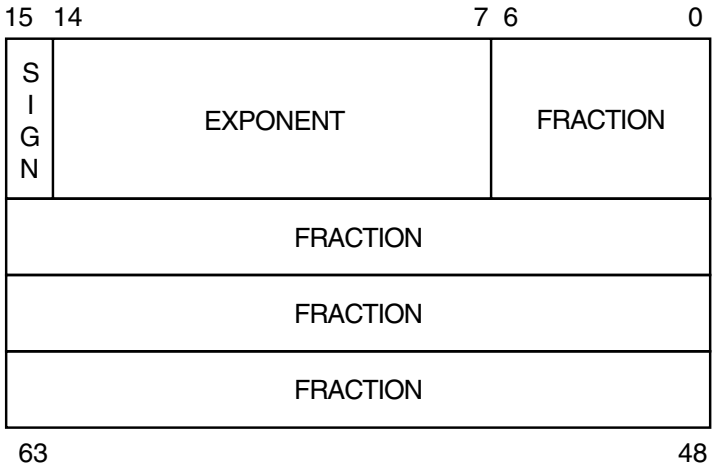
Figure 17–5 Single-Precision Real Number Format



17.2.2 DOUBLE Floating-Point Number Format (D_floating)

Double-precision real number format consists of eight contiguous bytes, starting on an arbitrary byte boundary. Bits are labeled from the right, 0 to 63, as in Figure 17–6. The form of a D_floating number is identical to the F_floating form, except for an additional 32 low-significance fraction bits. Within the fraction, bits increase in significance from 48 to 63, 32 to 47, 16 to 31, and 0 to 6. The exponent conventions and approximate range of values are the same for both D_floating and F_floating numbers. The precision of a D_floating number is approximately one part in 2^{55} (approximately 16 decimal digits).

Figure 17–6 Double-Precision Real Number Format



In Alpha BASIC, it is possible to lose three binary digits of precision in arithmetic operations when performing operations on D_floating double-precision floating-point data. For each arithmetic operation, the data is converted to G_floating first, the operation is performed in G_floating, and the result is converted back to D_floating when the operation is complete.

Note

Because most floating-point values cannot be represented exactly in binary, they are susceptible to rounding. I64 BASIC and the Itanium hardware use T_floating representation in place of D_floating representation. Alpha BASIC and the Alpha system hardware use G_floating representation in place of the D_floating representation. Thus, the behavior of floating-point computations and comparisons can be different from what you expect.

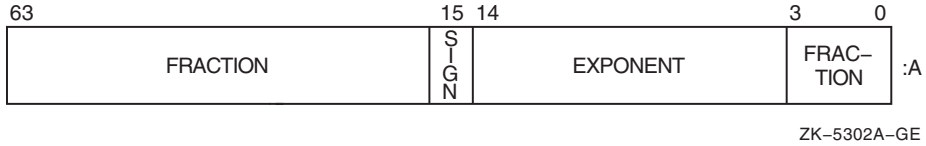
17.2.3 GFLOAT Floating-Point Number Format (G_floating)

The G_floating floating-point number format consists of eight contiguous bytes, starting on an arbitrary byte boundary, as shown in Figure 17–7. Bits are labeled from the right, 0 to 63. The form of a G_floating number is sign magnitude with bit 15 the sign bit, bits 14 to 4 an excess-1024 binary exponent, and bits 3 to 0 and 63 to 16 a normalized 53-bit fraction with the redundant most significant fraction bit not represented.

Within the fraction, bits increase in significance from 48 to 63, 32 to 47, 16 to 31, and 0 to 3. The 11-bit exponent field encodes the values 0 to 2047.

An exponent value of 0 together with a sign bit of 0 indicates that the G_floating number value is 0. Exponent values from 1 to 2047 indicate true binary exponents from -1023 to 1023. The value of a G_floating number is in the approximate range $.56 * 10^{-308}$ to $.9 * 10^{308}$; the precision is approximately one part in 2^{52} (approximately 15 decimal digits). Note that both double and G_floating formats require 8 bytes. The G_floating format provides a greater range, but less precision than double-precision format.

Figure 17–7 GFLOAT Floating-Point Number Format

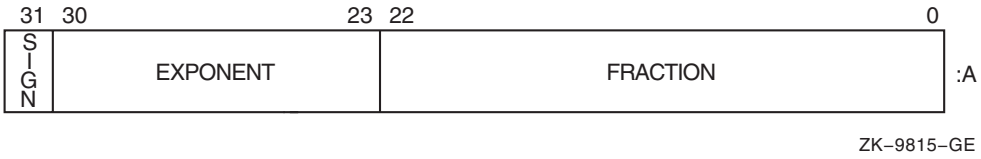


17.2.4 SFLOAT Floating-Point Number Format (S_floating)

The S_floating floating-point number format consists of four contiguous bytes, starting on an arbitrary byte boundary, as shown in Figure 17–8. Bits are labeled from the right, 0 to 31. The form of an S_floating number is sign magnitude with bit 31 the sign bit, bits 30 to 23 an excess-127 binary exponent, and bits 22 to 0 a normalized 24-bit fraction with the redundant most significant fraction bit not represented unless the exponent is 0. If the exponent is 0, a nonzero fraction represents an unnormalized 23-bit fraction.

An exponent value of 0 together with a fraction value of 0 and a sign bit of 0 indicates that the S_floating number value is 0. Exponent values from 1 to 254 indicate true binary exponents from -127 to 127. The value of an S_floating number is in the approximate range $1.175 * 10^{-38}$ to $3.402 * 10^{38}$; the precision is approximately one part in 2^{23} (approximately 7 decimal digits). Note that S_floating format provides approximately the same range and precision as F_floating format.

Figure 17–8 SFLOAT Floating-Point Number Format

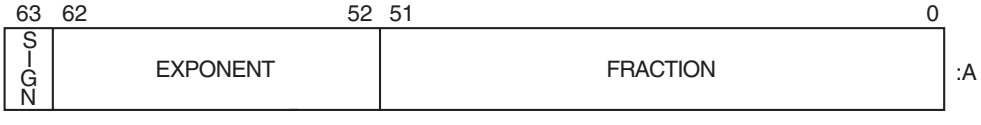


17.2.5 TFLOAT Floating-Point Number Format (T_floating)

The T_floating floating-point number format consists of eight contiguous bytes, starting on an arbitrary byte boundary, as shown in Figure 17–9. Bits are labeled from the right, 0 to 63. The form of a T_floating number is sign magnitude with bit 63 the sign bit, bits 62 to 52 an excess-1023 binary exponent, and bits 51 to 0 a normalized 53-bit fraction with the redundant most significant fraction bit not represented unless the exponent is 0. If the exponent is 0, a nonzero fraction represents an unnormalized 52-bit fraction.

An exponent value of 0 together with a fraction value of 0 and a sign bit of 0 indicates that the T_floating number value is 0. Exponent values from 1 to 2046 indicate true binary exponents from -1023 to 1023. The value of a T_floating number is in the approximate range $2.225 * 10^{-308}$ to $1.797 * 10^{308}$; the precision is approximately one part in 2^{52} (approximately 15 decimal digits). Note that T_floating format provides approximately the same range and precision as G_floating format.

Figure 17–9 TFLOAT Floating-Point Number Format

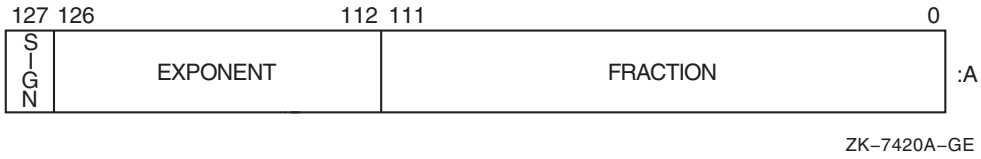


17.2.6 XFLOAT Floating-Point Number Format (X_floating)

The X_floating floating-point number format consists of sixteen contiguous bytes, starting on an arbitrary byte boundary, as shown in Figure 17–10. Bits are labeled from the right, 0 to 127. The form of an X_floating number is sign magnitude with bit 127 the sign bit, bits 126 to 112 an excess-16383 binary exponent, and bits 111 to 0 a normalized 113-bit fraction with the redundant most significant fraction bit not represented unless the exponent is 0. If the exponent is 0, a nonzero fraction represents an unnormalized 112-bit fraction.

An exponent value of 0 together with a fraction value of 0 and a sign bit of 0 indicates that the X_floating number value is 0. Exponent values from 1 to 32766 indicate true binary exponents from -16383 to 16383. The value of an X_floating number is in the approximate range $3.362 * 10^{-4932}$ to $1.189 * 10^{4932}$; the precision is approximately one part in 2^{112} (approximately 33 decimal digits). Note that X_floating format provides approximately the same range and precision as H_floating format.

Figure 17–10 XFLOAT Floating-Point Number Format



17.3 Packed Decimal Number Format

The DECIMAL data type is useful for storing numbers with a fixed decimal point. DECIMAL numbers are stored as a precise representation of the value stored within the constraints of the specified number of fractional digits.
A packed decimal string is a contiguous sequence of bytes in memory. The address A and length L are sufficient to specify a packed decimal string, but note that L is the number of digits, not bytes, in the string. Each byte of a packed decimal string is divided into two 4-bit fields (nibbles), each of which must contain decimal digits, except the low nibble of the last byte, which must contain a sign. The representation for the digits or signs is shown in the following table:

Digit or Sign	Decimal	Hexadecimal
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10,12,14, or 15	A,C,E, or F
–	11 or 13	B or D

Despite the options, the preferred sign representation is 12 for positive and 13 for negative. The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range 1 to 31. If the number of digits is odd, the digits and the sign fit into ((L/2) + 1) bytes; when the number of digits is even, an extra 0 digit must appear in the high nibble (bits 7 to 4) of the first byte.

The address A of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte.

Note that the decimal point is specified by the descriptor for the packed decimal string. See Section 17.6 for more information about packed decimal string descriptions.

17.4 String and Array Descriptor Format

A **descriptor** is an OpenVMS data structure that describes how to access data in memory. A descriptor can also pass information about a paramater with that parameter. The following sections describe the formats for fixed-length and dynamic string descriptors.

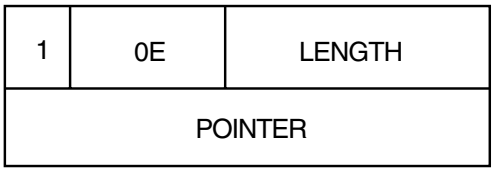
17.4.1 Fixed-Length String Descriptor Format

A fixed-length string descriptor consists of two longwords.

The first word of the first longword contains a value equal to the string's length. The third byte contains 14 (0E hexadecimal—the OpenVMS code describing an ASCII character string). The fourth byte contains 1.

The second longword is a pointer containing the address of the string's first byte. See Figure 17–11. For more information, see the *OpenVMS Calling Standard*.

Figure 17–11 Fixed-Length String Descriptor Format



ZK–5178–GE

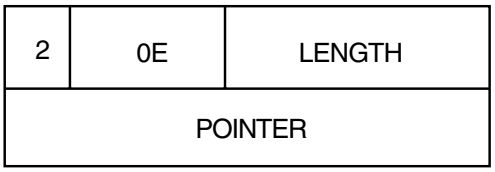
17.4.2 Dynamic String Descriptor Format

A dynamic string descriptor consists of two longwords.

The first word of the first longword contains a value equal to the string's length. The third byte contains 14 (0E hexadecimal—the OpenVMS code describing an ASCII character string). The fourth byte contains 2.

The second longword is a pointer containing the address of the string's first character. See Figure 17–12. For more information, see the *OpenVMS Calling Standard*.

Figure 17–12 Dynamic String Descriptor Format



ZK–5179–GE

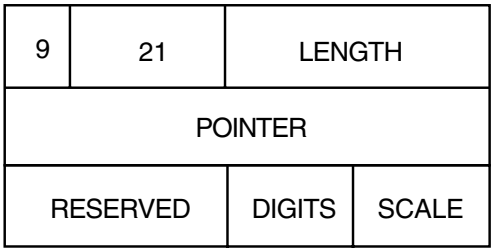
17.5 Array Descriptors

HP BASIC creates DSC$K_CLASS_NCA, a noncontiguous class array descriptor. For more information, see the *OpenVMS Calling Standard*.

17.6 Decimal Scalar String Descriptor (Packed Decimal String Descriptor)

A single descriptor form gives decimal size and scaling information for both scalar data and simple strings. See Figure 17–13 for more information.

Figure 17–13 Decimal Scalar String Descriptor



ZK–5181–GE

For packed decimal strings, the length field contains the number of 4-bit digits (not including the sign). The pointer field contains the address of the first byte in the packed decimal string. The scale field contains a signed power-of-ten multiplier to convert the internal form to the external form. For example, if the internal number is 123 and the scale field is +1, then the external number is 1230.

Part III

Using HP BASIC Features on OpenVMS Systems

Part III describes BASIC features available on OpenVMS systems including advanced file input and output, libraries and shareable images, and error messages.

18

Advanced File Input and Output

This chapter describes the more advanced I/O features available in HP BASIC. The following topics are presented:

- RMS I/O to ANSI magnetic tapes
- Device-specific I/O to magnetic tapes (including TK50 devices), disks, and unit record devices
- I/O to mailboxes
- Network I/O

When you do not specify a file name in the OPEN statement, the I/O you perform is said to be **device-specific**. This means that read and write operations (GET and PUT statements) are performed directly to or from the device. For example:

```
OPEN "MTA2:" FOR OUTPUT AS FILE #1
OPEN "MTA1:PARTS.DAT" FOR INPUT AS FILE #2, SEQUENTIAL
```

Because the file specification in the first line does not contain a file name, the OPEN statement opens the tape drive for device-specific I/O. The second line opens an ANSI-format tape file using RMS because a file name is part of the file specification.

The following sections describe both I/O to ANSI-format magnetic tapes and device-specific I/O to magnetic tape, unit record, and disk devices.

For more information about I/O to RMS disk files, see Chapter 13.

18.1 RMS I/O to Magnetic Tape

HP BASIC supports I/O to ANSI-formatted magnetic tapes. When performing I/O to ANSI-formatted magnetic tapes, you can read or write to only one file on a magnetic tape at a time, and the files are not available to other users. ANSI tape files are RMS sequential files.

18.1.1 Allocating and Mounting a Tape

You should allocate the tape unit to your process before starting file operations. For example:

```
$ ALLOCATE MT1:
```

This command assigns tape drive MT1: to your process. You must also set the tape density and label with the MOUNT command. Optionally, you can specify a logical name to assign to the device, in this case, TAPE.

```
$ MOUNT/DENSITY=1600 MT1: VOL001 TAPE
```

When mounting a TK50, you cannot specify a density.

If the records do not specify the size of the block (no value in HDR 2), specify the BLOCKSIZE as part of the MOUNT command. For example:

```
$ MOUNT/DENSITY=1600/BLOCKSIZE=128 MT1: VOL020 TAPE
```

Alternatively, you can use the $MOUNT system service to mount tapes.

18.1.2 Opening a Tape File for Output

To create and open a magnetic tape file for output, use the OPEN statement. The following statement opens the file PARTS.DAT and writes 256-byte records that are blocked four to a physical tape block of 1024 bytes:

```
OPEN "MT1:PARTS.DAT" FOR OUTPUT AS FILE #2%, SEQUENTIAL FIXED,  &
              RECORDSIZE 256%, BLOCKSIZE 4%
```

Specifying FIXED record format creates ANSI F format records. Specifying VARIABLE creates ANSI D format records. If you do not specify a record format, the default is VARIABLE.

Note

Every record in an ANSI D formatted file is prefixed by a 4-byte header giving the record length in decimal ASCII digits. The length includes the 4-byte header. HP BASIC adds the 4-byte header to the record size when calculating block size. The header is transparent to your program.

If you do not specify a block size, HP BASIC defaults to one record per block. For small records, this can be inefficient; the tape will contain many interrecord gaps.

18.1.3 Opening a Tape File for Input

To open an existing magnetic tape file, you also use the OPEN statement. For example, the following statement opens the file PAYROLL.DAT. If you do not specify a record size or a block size, HP BASIC defaults to the values in the header block. If you do not specify a record format, HP BASIC defaults to the format present in the header block (ANSI F or ANSI D). You must specify ACCESS READ if the tape is not write-enabled. For example:

```
100    OPEN "TAPE:PAYROLL.DAT" FOR INPUT AS FILE #4%
              ,ACCESS READ
```

18.1.4 Positioning a Tape

The NOREWIND statement positions the tape for reading and writing as follows:

- Specifying NOREWIND when you create a file positions the tape at the logical end-of-tape and leaves the unit open for writing. If you omit NOREWIND, you start writing at the beginning of the tape (BOT), logically deleting all subsequent files.
- Specifying NOREWIND when you open an existing file starts a search for the file at the current position. The search continues to the logical end-of-tape. If the record is not found, HP BASIC rewinds and continues the search until reaching the logical end-of-tape again. Omitting NOREWIND tells HP BASIC to rewind the tape and search for the file name until reaching the end-of-tape. In either case, you receive an error message if the file does not exist.

For example, the following statement opens PAYROL.DAT after advancing the tape to the logical end-of-tape. If you omit NOREWIND, the file opens at the beginning of the tape, logically deleting all subsequent files.

```
OPEN "MT1:PAYROL.DAT" FOR OUTPUT AS FILE #1% &
          ,ORGANIZATION SEQUENTIAL, NOREWIND
```

Note that you cannot specify REWIND; to avoid rewinding the tape, omit the NOREWIND keyword.

18.1.5 Writing Records to a File

The PUT statement writes sequential records to the file. The following program writes a record to the file. Successive PUT operations write successive records.

```
OPEN "MT0:TEST.DAT" FOR OUTPUT AS FILE #2, &
              SEQUENTIAL FIXED, RECORDSIZE 20%
B$ = ""
WHILE B$ <> "NO"
     LINPUT "Name"; A$
     MOVE TO #2, A$ = 20
     PUT #2
     LINPUT "Write another record"; B$
NEXT
CLOSE #2
END
```

Each PUT writes one record to the file. If your OPEN statement specifies a RECORDSIZE clause, the record buffer length equals RECORDSIZE or the map size. For example:

```
RECORDSIZE 60%
```

This clause specifies a record length and a record buffer size of 60 bytes. You can specify a record length from 18 to 8192 bytes. The default is 132 bytes.

If you specify a MAP clause and no RECORDSIZE clause, then the record size is the size of the map.

If you also specify BLOCKSIZE, the size of the buffer equals the value in BLOCKSIZE multiplied by the record size. For example:

```
RECORDSIZE 60%, BLOCKSIZE 4%
```

These clauses specify a logical record length of 60 bytes and a physical tape record size of 240 bytes (60 * 4). You specify BLOCKSIZE as an integer number of records. RMS rounds the resulting value to the next multiple of four. The total I/O buffer length cannot exceed 8192 bytes. The default is a buffer (tape block) containing one record.

To write true variable-length records, use the COUNT clause with the PUT statement to specify the number of bytes of data written to the file. Without COUNT, all records equal the length specified by the RECORDSIZE clause when you opened the file.

18.1.6 Reading Records from a File

The GET statement reads one logical record into the buffer. In the following example, the first GET reads a group of four records (a total of 80 bytes) from the file on channel #5 and transfers the first 20 bytes to the record buffers. Successive GET operations read 20 byte records to the record buffer performing an I/O to the tape every 4 records.

```
OPEN "MT0:TEST.DAT" FOR INPUT AS FILE #5%, &
       ORGANIZATION SEQUENTIAL FIXED, RECORDSIZE 20%, &
       BLOCKSIZE 4%, ACCESS READ
B$ = ""
WHILE B$ <> "NO"
      GET #5
      MOVE FROM #5, A$ = 20
      PRINT A$
      LINPUT "Do you want another record"; B$
NEXT
CLOSE #5
END
```

18.1.7 Controlling Tape Output Format

Magnetic tape physical records range from 18 to 8192 bytes. With RMS tapes, you can optionally specify this size in the BLOCKSIZE clause as a positive integer indicating the number of records in each block. HP BASIC then calculates the actual size in bytes. Thus, a fixed-length file on tape with 126 byte records can have a block size from 1 to 64, inclusive. The default is 126 bytes (one record per block).

In the following example of an OPEN statement, the RECORDSIZE clause defines the size of the records in the file as 90 bytes, and BLOCKSIZE defines the size of a block as 12 records (1080 bytes). Thus, your program contains an I/O buffer of 1080 bytes. Each physical read or write operation moves 1080 bytes of data between the tape and this buffer. Every twelfth GET or PUT operation causes a physical read or write. The next eleven GET or PUT operations only move data into or out of the I/O buffer. Specifying a block size larger than the default can reduce overhead by eliminating some physical reading and writing to the tape. In addition, specifying a large block size conserves space on the tape by reducing the number of interrecord gaps (IRGs). In the example, a block size of 12 saves time by accessing the tape only after every twelfth record operation.

```
OPEN "MT0:[SMITH]TEST.SEQ" FOR OUTPUT AS FILE #12% &
               ,ORGANIZATION SEQUENTIAL FIXED, RECORDSIZE 90% &
               ,BLOCKSIZE 12%
```

Through RMS, HP BASIC controls the blocking and deblocking of records. RMS checks each PUT operation to see if the specified record fits in the tape block. If it does not, RMS fills the rest of the block with circumflexes (blanks) and starts the record in a new block. Records cannot span blocks in magnetic tape files.

When you read blocks of records, your program can issue successive GET statements until it locates the fields of the record you want. The following program finds and displays a record on the terminal. You can invoke the RECOUNT function to determine how many bytes were read in the GET operation.

```
MAP  (XXX)  NA.ME$ = 5%, address$ = 20%

OPEN "MTO:FILE.DAT" FOR INPUT AS FILE #4%, &
           SEQUENTIAL FIXED, MAP XXX, ACCESS READ

NA.ME$ = ""
GET #4 UNTIL NA.ME$ - "JONES"
PRINT NA.ME$;  "LIVES AT "; address$

CLOSE #4

END
```

18.1.8 Rewinding a Tape

With the RESTORE # statement, you can rewind the tape to the start of the currently open file. For example:

```
OPEN  "MTO:FTF.DAT" FOR INPUT AS FILE #2%, ACCESS READ
GET #2%
   .
   .
   .
RESTORE #2%
GET #2%
```

You cannot rewind past the beginning of the currently open file.

18.1.9 Closing a File

The CLOSE statement ends I/O to the file. The following statement ends input and output to the file open on channel #6:

```
CLOSE #6%
```

If you opened the file with ACCESS READ, CLOSE has no further effect. If you opened the file without specifying ACCESS READ and the tape is not write-locked (that is, if the plastic write ring is in place), HP BASIC does the following:

- Writes the file trailer labels and two end-of-file marks following the last record
- Backspaces over the last end-of-file mark

HP BASIC does not rewind the tape.

18.2 Device-Specific I/O

Device-specific I/O lets you perform I/O directly to a device. The following sections describe device-specific I/O to unit record devices, tapes, and disks.

18.2.1 Device-Specific I/O to Unit Record Devices

You perform device-specific I/O to unit record devices by using only the device name in the OPEN statement file specification. You should allocate the device at DCL command level before reading or writing to the device. For example, this command allocates a card reader:

```
$ ALLOCATE CR1:
```

Once the device is allocated, you can read records from it. For example:

```
MAP  (DNG)  A% = 80%
OPEN  "CR1:"  FOR INPUT AS FILE #1%, ACCESS READ, MAP DNG
GET #1%
```

HP BASIC treats the device as a file, and data is read from the card reader as a series of fixed-length records.

18.2.2 Device-Specific I/O to Magnetic Tape Devices

When performing device-specific I/O to a tape drive, you open the physical device and transfer data between the tape and your program. GET and PUT statements perform read and write operations. UPDATE and DELETE statements are invalid when you perform device-specific I/O.

18.2.2.1 Allocating and Mounting a Tape

You must allocate the tape unit to your process before starting file operations. The following command line assigns tape drive MT1: to your process:

```
$ ALLOCATE MT1:
```

Use the DCL command MOUNT and the /FOREIGN qualifier to mount the tape. For example:

```
$ MOUNT/FOREIGN MT1:
```

If your program needs a blocksize other than 512 bytes, or a particular tape density, specify these characteristics with the MOUNT command as well. For example:

```
$ MOUNT/FOREIGN/BLOCKSIZE=1024/DENSITY=1600 MT1:
```

When reading a foreign tape, you must make sure the /BLOCKSIZE qualifier has a value at least as large as the largest record on the tape.

18.2.2.2 Opening a Tape File for Output

To create and open the magnetic tape for output, you use the OPEN statement. The following statement opens tape drive MT1: for writing. It is important to use the SEQUENTIAL VARIABLE clause unless the records are fixed. In contrast to ANSI tape processing, RMS does not write record length headers or variable-length records to foreign tapes. If you specify SEQUENTIAL VARIABLE, you should have some way to determine where records begin and end.

```
OPEN  "MT1:"  FOR OUTPUT AS FILE #1%,         &
        ORGANIZATION SEQUENTIAL VARIABLE
```

18.2.2.3 Opening a Tape File for Input

To access a tape with existing data, you also use the OPEN statement. For example, the following statement opens the tape unit MT2:.

```
OPEN  "MT2:"  AS FILE #2%
```

Depending on how you access records, there are two ways to open a foreign magnetic tape. If your program uses dynamic buffering and MOVE statements, open the file with no RECORDSIZE clause. RMS will provide the correct buffer size for HP BASIC. Do not specify a BLOCKSIZE value or ORGANIZATION clause with the OPEN statement.

If your program uses MAP and REMAP statements, but you do not know how long the records are, specify a MAP that is as large as the value you specified for the BLOCKSIZE qualifier when mounting the tape. Do not specify a BLOCKSIZE value or ORGANIZATION clause with the OPEN statement.

When processing records, each GET operation will read one physical record whose size is returned in RECOUNT. If you are using a map only, the first *n* bytes (*n* is the value returned in RECOUNT) are valid.

18.2.2.4 Writing Records to a File

The PUT statement writes records to the file in sequential order. For example:

```
OPEN "MT0:" FOR OUTPUT AS FILE #9%, &
          SEQUENTIAL VARIABLE
INPUT "NAME";NA.ME$
MOVE TO #9%, NA.ME$
PUT #9%
```

The last line writes the contents of the record buffer to the device. Successive PUT operations write successive records.

The default record length (and, therefore, the size of the buffer) is 132 bytes. The RECORDSIZE attribute causes HP BASIC to read or write records of a specified length. For example, the following statement opens tape unit MT0: and specifies records of 900 characters. You must specify an even integer larger than or equal to 18. If you specify a buffer length less than 18, HP BASIC signals an error. If you try to write a record longer than the buffer, HP BASIC signals the error "Size of record invalid" (ERR=156).

```
OPEN "MT0:" FOR INPUT AS FILE #1%, RECORDSIZE 900%
```

To write records shorter than the buffer, include the COUNT clause with the PUT statement. The following statement writes a 56-character record to the file open on channel #6. If you do not specify COUNT, HP BASIC writes a full buffer. You can specify a minimum count of 18, and a maximum count equal to the buffer size. When writing records to a foreign magnetic tape, neither HP BASIC nor RMS prefixes the records with any count bytes.

```
PUT #6%, COUNT 56%
```

18.2.2.5 Reading Records from a File

The GET statement reads records into the buffer. The following program reads a record into the buffer, prints a string field, and rewinds the file before closing. Successive GET operations read successive records. HP BASIC signals the error "End of file on device" (ERR=11) if you encounter a tape mark during a GET operation. If you trap this error and continue, you can skip over any tape marks. The system variable RECOUNT is set to the number of bytes transferred after each GET operation.

```
OPEN "MT1:" FOR INPUT AS FILE #1%, ACCESS READ
GET #1%
MOVE FROM #1%, A$ = RECOUNT
PRINT A$
RESTORE #1%
CLOSE #1%
```

18.2.2.6 Rewinding a Tape

When you mount a magnetic tape, the system will position the tape at the load point (BOT). Your program can rewind the tape during program execution with the RESTORE statement. For example:

```
OPEN "MT1:" FOR OUTPUT AS FILE #2%, ACCESS READ
   .
   .
   .
PUT #2%
RESTORE #2%
INPUT "NEXT RECORD"; NXTRECBB%
```

If you rewind a tape opened without ACCESS READ before closing it, you erase all data written before the RESTORE operation.

18.2.2.7 Closing a Tape

The CLOSE statement ends I/O to the tape. For example, the following statement ends input and output to the tape open on channel #12.

```
CLOSE #12%
```

If you opened the file with ACCESS READ, CLOSE has no further effect. If you opened the file without specifying ACCESS READ and the tape is not write-locked (that is, if the plastic write ring is in place), HP BASIC does the following:

- Writes file trailer labels and two end-of-file marks following the last record
- Backspaces over the last end-of-file mark

The tape is not rewound unless you specified RESTORE in your program.

18.2.3 Device-Specific I/O to Disks

When performing device-specific I/O to disks, you write and read data with PUT and GET statements. The data must fit in 512-byte blocks, and you must do your own blocking and deblocking with MAP/REMAP or MOVE statements. Note that, when accessing disks with device-specific I/O operations, you are performing logical I/O. Because of this, you should be careful not to overwrite block number zero, which is often the disk's boot block. You must have LOG_ IO privileges to perform these operations.

The following sections describe device-specific I/O to disks.

18.2.3.1 Assigning and Mounting a Disk

You must allocate a disk unit to your process before starting operations. The following command line assigns disk DUA3: to your process:

```
$ ALLOCATE DUA3:
```

When you perform I/O directly to a disk, you must mount the disk with the MOUNT command before accessing it. For example:

```
$ MOUNT/FOREIGN DUA3:
```

You can then open the disk for input or output.

18.2.3.2 Opening a Disk File for Output

To create and open the disk file, you use the OPEN statement. For example:

```
OPEN "DUA3:" FOR OUTPUT AS FILE #2%, SEQUENTIAL FIXED, &
       RECORDSIZE=512
```

You can then write data to the disk.

The record size determined by the MAP or RECORDSIZE clause must be an integer multiple of 512 bytes.

18.2.3.3 Opening a Disk File for Input

To open an existing disk file, you also use the OPEN statement. For example:

```
OPEN "DUA1:" FOR INPUT AS FILE #4%, SEQUENTIAL FIXED, &
       RECORDSIZE=512
```

You can then read data from the disk.

The record size determined by the MAP or RECORDSIZE clause must be an integer multiple of 512 bytes. The default is 512.

Specify ACCESS READ in the OPEN statement if you only plan to read from the disk.

18.2.3.4 Writing Records to a Disk File

You write data by defining a record buffer and writing the data to the file with PUT statements. The following program writes eight 64 byte records into each 512-byte block on the disk. When your program fills one block, writing continues in the next. The FILL field in the MOVE statement positions the data in the block.

```
INPUT "HOW MANY RECORDS TO WRITE"; J%
OPEN "DBB2: FOR OUTPUT AS FILE #2%, SEQUENTIAL FIXED, &
      RECORDSIZE=512
FOR K% = 1% TO J%
   FOR I% = 0% TO 7%
        INPUT "NAME OF BOOK"; BOOK_NAME$
        INPUT "RETRIEVAL NUMBER"; RET_NUM%
        INPUT "SUBJECT AREA"; SUBJ$
        MOVE TO #2%, FILL$ = I% * 64%, BOOK_NAME$, RET_NUM%, SUBJ$
   NEXT I%
PUT #2%
NEXT K%
CLOSE #2
```

When you write records, HP BASIC does not prefix the records with any count bytes.

18.2.3.5 Reading Records from a Disk File

You read data by defining a record buffer and reading the data from the device with GET statements. After the data has been retrieved with a GET statement you can deblock the data with MOVE or REMAP statements.

In the following example, each disk block contains twelve 40-byte records. Each record contains a 32-byte string, a 4-byte SINGLE number, and a 4-byte LONG integer. After each GET operation, the FOR...NEXT loop uses the REMAP statement to redefine the position of the variables in the record. At the end of the file, the program closes the file. See Chapter 7 and the *HP BASIC for OpenVMS Reference Manual* for more information about the MAP, MAP DYNAMIC, and REMAP statements.

```
MAP (SAM) FILL$ = 512
MAP DYNAMIC (SAM) STRING PRT_ID, SINGLE MAFLD, LONG ADIR_OLDN
OPEN "DUA1:" FOR INPUT AS FILE #2%, SEQUENTIAL FIXED, &
            ACCESS READ, MAP SAM
WHEN ERROR USE err_hand
WHILE 1% = 1%
     GET #2%
     FOR I% = 0% TO 11%
         REMAP (SAM) STRING FILL(I% * 40%), PRT_ID = 32, MAFLD, ADIR_OLDN
         PRINT PRT_ID, MAFLD, ADIR_OLDN
     NEXT I%
NEXT
END WHEN
HANDLER err_hand
  IF ERR <> 11%
    THEN
      EXIT HANDLER
  END IF
END HANDLER
CLOSE #2%
END
```

18.3 I/O to Mailboxes

A **mailbox** is a record I/O device that passes data from one process to another. You can use a valid mailbox name as a file name, and treat that mailbox as a normal record file. You must have TMPMBX or PRMMBX privilege to create mailboxes. Mailboxes are created and deleted by system services. For more information about using system services in HP BASIC programs, see Chapter 19.

Use the EXTERNAL statement to define the SYS$CREMBX system service that creates the mailbox. In HP BASIC programs, you create mailboxes by invoking SYS$CREMBX as a function passing either a channel argument and a string literal or a logical name for the mailbox. For example:

```
EXTERNAL INTEGER FUNCTION SYS$CREMBX
SYS$STATUS% = SYS$CREMBX(,CHAN%,,,,,"CONFIRMATION_MBX")
```

If you supply a logical name for the mailbox, be sure that it is in uppercase letters. Once you create the mailbox, you can use it as a logical file name.

The following two examples, when executed on two separate processes, allow you to send and receive data from one process to another.

Example 1

```
DECLARE STRING passenger_name, Confirm_msg
OPEN "CONFIRMATION_MBX" AS FILE #1%
INPUT "WHAT IS THE PASSENGER NAME"; passenger_name
PRINT #1%, passenger_name
LINPUT #1%, confirm_msg
PRINT confirm_msg
END
```

Example 2

```
MAP (res) STRING passenger_name = 32%
DECLARE WORD mbx_chan, LONG sys_status
EXTERNAL LONG FUNCTION sys$crembx (LONG, WORD, LONG, LONG,    &
                                   LONG, LONG, STRING)
WHEN ERROR USE err_trap
sys_status = sys$crembx ( ,mbx_chan,,,,,"CONFIRMATION_MBX")
OPEN "CONFIRMATION_MBX" FOR INPUT AS FILE #1%
LINPUT #1%, passenger_name
OPEN "RESER.LST" FOR INPUT AS FILE #2%,            &
        ORGANIZATION INDEXED, MAP RES, ACCESS READ            &
        PRIMARY passenger_name
FIND #2%, KEY #0% EQ passenger_name
RECEIVING.MSG$ = "Passenger reservation confirmed"
PRINT #1%, RECEIVING.MSG$
END WHEN
HANDLER err_trap
    IF (ERR = 155)
       THEN
       RECEIVING.MSG$ = "Reservation does not exist"
       ELSE
       EXIT HANDLER
    END IF
END HANDLER
CLOSE #2%, #1%
END PROGRAM
```

Example 1 requests a passenger name and sends it to the mailbox.

Example 2 looks up the name in an indexed file. If the passenger name exists, Example 2 writes the confirmation message to the mailbox. If the passenger name does not exist, the error handler writes an alternate message. Example 1 then reads the mailbox and returns the result.

HP BASIC treats the mailbox as a sequential file. You write to the file with the PRINT # or PUT statement, and read it with the INPUT #, LINPUT #, or GET statement.

When either program closes the mailbox, the other program receives an end-of-file error message when it attempts to read the mailbox.

Note

All mailbox operations are synchronous. Control does not pass back from a mailbox operation, such as a PUT, to your program until the other program completes the corresponding operation, such as a GET.

18.4 Network I/O

If your system supports DECnet for OpenVMS VAX facilities, and your computer is one of the nodes in a DECnet for OpenVMS VAX, you can communicate with other nodes in the network with HP BASIC program statements. HP BASIC lets you do the following:

- Read and write files on a remote node as you do files on your own system (remote file access)
- Exchange data with a process executing at a remote location (task-to-task communication)

18.4.1 Remote File Access

To write or read files at a remote site, include the node name as part of the file specification. For example:

```
OPEN "WESTON::DUA1:[HOLT]TEST.DAT;2" FOR INPUT AS FILE #2%
```

You can also assign a logical name to the file specification, and use that logical name in all file I/O.

Note

You need NETMBX privileges to access files at a remote node.

If the account at the remote site requires a username and password, include this access string in the file specification. You do this by enclosing the access string in quotation marks and placing it between the node name and the double colon. The following file specification accesses the account [HOLT.TMP] on node WESTON by giving the username HOLT and the password PASWRD. After accessing the file, your HP BASIC program can read and write records as if the file were in your account.

```
OPEN 'WESTON"HOLT PASWRD"::DUA0:[HOLT.TMP]INDEXU.DAT;4' &
          FOR INPUT AS FILE #1%, INDEXED, PRIMARY TEXT$
```

Do not use the CONNECT clause when opening a file on a remote node or HP BASIC will signal the error "Cannot open file" (ERR=162).

18.4.2 Task-to-Task Communication

HP BASIC supports task-to-task communication if your account has NETMBX privileges.

Follow these steps for task-to-task communication:

1. Establish a command file at the remote site to execute the program you want. The program must be in executable image format. For example, you can create the file MARG.COM at the remote site. MARG.COM contains a line to run an image (in this case, COPYT.EXE).

   ```
   $ RUN COPYT
   ```

   The OPEN statements in the programs at both nodes must specify the same file attributes.

2. Start task-to-task communication by accessing the command file at the remote site. For example, a program at the local node could contain the following line:

   ```
   OPEN 'WESTON::"TASK = MARG"' AS FILE #1%, SEQUENTIAL
   ```

3. The system then assigns the logical name SYS$NET to the program at the local node. At the remote node, the program (COPYT.EXE) must use this logical for all operations. For example:

   ```
   OPEN 'SYS$NET' FOR INPUT AS FILE #1%, SEQUENTIAL
   ```

4. The two programs can then exchange messages. The programs must have a complementary series of send/receive statements.

```
!Local Program
MAP (SJK) MSG$ = 32%
OPEN 'WESTON"DAVIS PSWRD"::"TASK = MARG"' &
        FOR OUTPUT AS FILE #1%, SEQUENTIAL, MAP SJK
LINPUT "WHAT IS THE CUSTOMER NAME"; MSG$
PUT #1%
GET #1%
PRINT MSG$
CLOSE #1%
END

!Remote Node Program
   .
   .
   .
10   MAP (SJK) MSG$ = 32%
     MAP (FIL) NAME$ = 32%, RESERVATION$ = 64%
     OPEN 'SYS$NET' FOR INPUT AS FILE #1%, SEQUENTIAL, &
            MAP SJK
     OPEN 'RESER.DAT'FOR INPUT AS FILE #2%,  &
              INDEXED FIXED, PRIMARY NAME$, MAP FIL
     GET #1%
     MSG$ = "NAME CONFIRMED"
     WHEN ERROR IN
100  FIND #2%, KEY 0% EQ MSG$
     USE
        IF ERR = 153
          THEN
             MSG$ = "ERROR IN NAME"
          ELSE
             EXIT HANDLER
        END IF
     END WHEN

     PUT #1%
   .
   .
   .
     CLOSE #2%, 1%
     END
```

The task-to-task communication ends when the files are closed.

See the *DECnet for OpenVMS Networking Manual* and the *HP OpenVMS System Manager's Manual* for more information.

18.4.3 Accessing a VAX Rdb/VMS Database

If you have purchased a VAX Rdb/VMS development license, you can store and access data in a VAX Rdb/VMS database from a HP BASIC program. To do this, you embed RDO statements in your HP BASIC program. Each line of an RDO statement must be preceded by the Rdb/VMS statement flag (&RDB&). HP BASIC line numbers cannot be included in any RDO statement line. You then precompile your program with the Rdb/VMS precompiler. The precompiler translates the RDO statements into BASIC statements that make direct calls to Rdb/VMS.

19

Using BASIC in the Common Language Environment

This chapter shows you how to call the following:

- External routines written in other OpenVMS languages
- OpenVMS Run-Time Library routines
- OpenVMS system services

The terms routine, procedure, and function are used throughout this chapter. A **routine** is a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name), and may or may not have an argument list. Procedures and functions are specific types of routines: a **procedure** is a routine that does not return a value, while a **function** is a routine that returns a value by assigning that value to the function's identifier.

System routines are prewritten OpenVMS routines that perform common tasks such as finding the square root of a number or allocating virtual memory. You can call any system routine from HP BASIC provided that the data structures necessary for that routine are supported. The system routines used most often are OpenVMS Run-Time Library routines and system services. System routines, which are discussed later in this chapter, are documented in detail in the *OpenVMS Run-Time Library Routines Volume* and the *HP OpenVMS System Services Reference Manual*.

19.1 Specifying Parameter-Passing Mechanisms

When you pass data between routines that are not written in the same language, you have to specify how you want that data to be represented and interpreted. You do this by specifying a **parameter-passing mechanism**. The general parameter-passing mechanisms and their keywords in HP BASIC are as follows:

- By reference—BY REF

- By descriptor—BY DESC
- By value—BY VALUE

The following sections outline each of these parameter-passing mechanisms in more detail.

19.1.1 Passing Parameters by Reference

When you pass a parameter by reference, HP BASIC passes the address at which the actual parameter value is stored. In other words, your routine has access to the parameter's storage address; therefore, you can manipulate and change the value of this parameter. Any changes that you make to the value of the parameter in your routine are reflected in the calling routine as well.

19.1.2 Passing Parameters by Descriptor

A **descriptor** is a data structure that contains the address of a parameter, along with other information such as the parameter's data type and size. When you pass a parameter by descriptor, the HP BASIC compiler passes the address of a descriptor to the called routine. You usually use descriptors to pass parameters that have unknown lengths, such as the following:

- Character strings
- Arrays
- Compound data structures

Like parameters passed by reference, any change made to the value of a parameter passed by descriptor is reflected in the calling routine.

19.1.3 Passing Parameters by Value

When you pass a parameter by value, you pass a copy of the parameter value to the routine instead of passing its address. Because the actual value of the parameter is passed, the routine does not have access to the storage location of the parameter; therefore, any changes that you make to the parameter value in the routine do not affect the value of that parameter in the calling routine.

HP BASIC allows actual and formal parameters to be passed by value.

19.1.4 HP BASIC Default Parameter-Passing Mechanisms

There are default parameter-passing mechanisms established for every data type you can use with HP BASIC. Table 19–1 shows which HP BASIC data types you can use with each parameter-passing mechanism.

Table 19–1 Valid Parameter-Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
Integer and Real Data			
Variables	Yes	Yes[1]	Yes
Constants	Yes	Local copy[1]	Local copy
Expressions	Yes	Local copy[1]	Local copy
Elements of a nonvirtual array	Yes	Yes[1]	Yes
Virtual array elements	Yes	Local copy[1]	Local copy
Nonvirtual entire array	No	Yes	Yes[1]
Virtual entire array	No	No	No
Packed Decimal Data			
Variables	No	Yes[1]	Yes
Constants	No	Local copy[1]	Local copy
Expressions	No	Local copy[1]	Local copy
Nonvirtual array elements	No	Yes[1]	Yes
Virtual array elements	No	Local copy[1]	Local copy

[1]Specifies the default parameter-passing mechanism.

(continued on next page)

Table 19–1 (Cont.) Valid Parameter-Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
Packed Decimal Data			
Nonvirtual entire arrays	No	Yes	Yes[1]
Virtual entire arrays	No	No	No
String Data			
Variables	No	Yes	Yes[1]
Constants	No	Local copy	Local copy[1]
Expressions	No	Local copy	Local copy[1]
Nonvirtual array elements	No	Yes	Yes[1]
Virtual array elements	No	Local copy	Local copy[1]
Nonvirtual entire arrays	No	Yes	Yes[1]
Virtual entire arrays	No	No	No
Other Parameters			
RECORD variables	No	Yes[1]	No
RFA variables	No	Yes[1]	No

[1]Specifies the default parameter-passing mechanism.

19.1.5 Creating Local Copies

If a parameter is an expression, function, or virtual array element, then it is not possible to pass the parameter's address. In these cases, HP BASIC makes a local copy of the parameter's value and passes this local copy by reference.

You can force HP BASIC to make a local copy of any parameter by enclosing the parameter in parentheses. Forcing HP BASIC to make a local copy is a useful technique because you make it impossible for the subprogram to modify the actual parameter. In the following example, when variable *A* is printed in

the main program, the value is zero because the variable *A* is not modifiable by the subprogram:

```
DECLARE LONG A
CALL SUB1 ((A))
PRINT A
END

SUB SUB1 (LONG B)
B = 3
END SUB
```

Output

```
 0
```

By removing the extra parentheses from *A*, you allow the subprogram to modify the parameter.

```
DECLARE LONG A
CALL SUB1 (A)
PRINT A
END

SUB SUB1 (LONG B)
B = 3
END SUB
```

Output

```
 3
```

19.1.6 Passing Arrays

In HP BASIC, if a subprogram or function declares an array in its parameter list, the calling program must pass an array. Passing a null parameter instead would cause the program to fail with a memory access violation.

19.2 Calling External Routines

Most of the steps of calling external routines are the same whether you are calling an external routine written in HP BASIC, an external routine written in some other language, a system service, or a OpenVMS Run-Time Library routine. The following sections outline the procedure for calling non-BASIC external routines. For information about calling BASIC routines, see Chapter 12.

19.2.1 Determining the Type of Call

Before you call an external routine, you must determine whether the call to the routine should be a function call or a procedure call. You should call a routine as a function if it returns any type of value. If the routine does not return a value, you should call it as a procedure.

19.2.2 Declaring an External Routine and Its Arguments

To call an external routine or system routine you need to declare it as an external procedure or function and to declare the names, data types, and passing mechanisms for the arguments. Arguments can be either required or optional.

You should include the following information in a routine declaration:

- The name of the external routine
- The data types of all the routine parameters
- The passing mechanisms for all the routine parameters, provided that the routine is not written in HP BASIC

When you declare an external routine, use the EXTERNAL statement. This allows you to specify the data types and parameter-passing mechanisms only once.

In the following example, the EXTERNAL statement declares *cobsub* as an external subprogram with two parameters—a LONG integer and a string both passed by reference:

```
EXTERNAL SUB cobsub (LONG BY REF, STRING BY REF)
```

With the EXTERNAL statement, HP BASIC allows you to specify that particular parameters do not have to conform to specific data types and that all parameters past a certain point are optional. A parameter declared as ANY indicates that any data type can appear in the parameter position. In the following example, the EXTERNAL statement declares a SUB subprogram named *allocate*. This subprogram has three parameters: one LONG integer, and two that can be of any HP BASIC data type.

```
EXTERNAL SUB allocate(LONG, ANY,)
```

A parameter declared as OPTIONAL indicates that all following parameters are optional. You can have both required and optional parameters. The required parameters, however, must appear before the OPTIONAL keyword because all parameters following it are considered optional.

In the following example, the EXTERNAL statement declares the Run-Time Library routine LIB$LOOKUP_KEY. The keyword OPTIONAL is specified to indicate that the last three parameters can be optional.

```
EXTERNAL LONG FUNCTION LIB$LOOKUP_KEY(STRING, LONG, OPTIONAL LONG, STRING, INTEGER)
```

For more information about using the EXTERNAL statement, see the *HP BASIC for OpenVMS Reference Manual*.

19.2.3 Calling the Routine

Once you have declared an external routine, you can invoke it. To invoke a procedure, you use the CALL statement. To invoke a function, you use the function name in an expression. You must specify the name of the routine being invoked and all parameters required for that routine. Make sure the data types and passing mechanisms for the actual parameters you are passing match those you declared earlier, and those declared in the routine.

If you do not want to specify a value for a required parameter, you can pass a null argument by inserting a comma as a placeholder in the argument list. If you are passing a parameter using a mechanism other than the default passing mechanism for that data type, you must specify the passing mechanism in the CALL statement or the function invocation.

The following example shows you how to call the external subprogram *allocate* declared in Section 19.2.2. When *allocate* is called, it is called as a procedure. The first parameter must always be a valid LONG INTEGER value; the second and third parameters can be of any valid HP BASIC data type.

```
EXTERNAL SUB allocate(LONG, ANY,)
   .
   .
   .
CALL allocate (entity%, a$, 1%)
```

This next example shows you how to call the Run-Time Library routine LIB$LOOKUP_KEY declared in Section 19.2.2. When the routine LIB$LOOKUP_KEY is called, it is invoked as a function. The first two parameters are required; all remaining parameters are optional.

```
EXTERNAL LONG FUNCTION LIB$LOOKUP_KEY(STRING, LONG, OPTIONAL LONG, STRING, INTEGER)
   .
   .
   .
ret_status% = LIB$LOOKUP_KEY(value$, point%)
```

Note that if the actual parameter's data type in the CALL statement does not match that specified in the EXTERNAL statement, HP BASIC reports the compile-time informational message "Mode for parameter of routine changed to match declaration." This tells you that HP BASIC has made a local copy of the value of the parameter, and that this local copy has the data type specified in the EXTERNAL declaration. HP BASIC warns you of this because the change means that the parameter can no longer be modified by the subprogram. If HP BASIC cannot convert the data type, HP BASIC signals the error "Mode for parameter of routine not as declared."

The routine being called receives control, executes, and then returns control to the calling routine at the next statement after the CALL statement or function invocation.

HP BASIC provides the built-in function LOC to allow you to access the address of a named external function. This is especially useful when passing the address of a callback or AST routine to an external subprogram. In the following example, the address of the function *compare* is passed to the subprogram *come_back_now* using the LOC function:

```
EXTERNAL LONG FUNCTION compare (LONG, LONG)
EXTERNAL SUB come_back_now (LONG BY VALUE)
CALL come_back_now (LOC(compare) BY VALUE)
```

19.3 Calling HP BASIC Subprograms from Other Languages

When you call a HP BASIC subprogram from another language, there are some additional considerations that you should be aware of. For example, although HP BASIC conforms to the OpenVMS Calling Standard, you should specify explicit passing mechanisms when calling a routine written in another language. The default passing mechanisms of BASIC may not match what the procedure expects. In the following section, FORTRAN refers to VAX FORTRAN and HP Fortran.

FORTRAN passes and receives numeric data by reference; only the default parameter-passing mechanisms are required for passing numeric data back and forth between FORTRAN and HP BASIC programs.

Both HP BASIC and FORTRAN pass strings by descriptor. However, FORTRAN subprograms cannot change the length of strings passed to them. Therefore, if you pass a string to a FORTRAN subprogram, you must make sure that the string is long enough to receive the result. You do this in one of two ways:

- Pre-extend the string. Set the string variable equal to SPACE$*(n)*, where *n* is large enough to receive the result.

- Define the string as fixed-length. Name the string in a COMMON or MAP statement.

Because the length of the returned string does not change, it is either padded with spaces or truncated.

To pass an array to a FORTRAN subprogram, you must specify BY REF.

Note that FORTRAN arrays are one-based, while HP BASIC arrays are zero-based by default. For example, in FORTRAN the array *Two_D*(5,3) represents a 5 by 3 matrix, while in HP BASIC the array *Two_d*(5,3) represents a 6 by 4 matrix. You can adjust your array bounds in HP BASIC by using the keyword TO when defining the array bounds. For more information about array bounds, see Chapter 6.

When passing two-dimensional arrays as parameters, keep in mind that FORTRAN addresses array elements in column major order, while BASIC refers to array elements in row major order. That is, FORTRAN arrays are of the form *Fortran_array*(column,row), while HP BASIC array elements are addressed as *Basic_array*(row,column). The FORTRAN array *Grid*(x,y) is therefore referred to as *GRID*(y,x) in HP BASIC. You should reverse references to array elements when passing arrays between HP BASIC and FORTRAN program modules. You can do this in one of two ways:

- Reverse array bounds in parameter lists
- Switch row and column variables within loops in your program module

Example 19–1 shows a HP BASIC program that passes a two-dimensional array to a FORTRAN subprogram. The FORTRAN subprogram is shown in Example 19–2.

Example 19–1 BASIC Main Program

```
PROGRAM call_fortran
    ! The BASIC main program prints the array before
    ! calling the subroutine
        EXTERNAL SUB forsub (WORD DIM(,) BY REF)
        DIM WORD array_x(1 TO 10, 1 TO 5)
        FOR column = 1 TO 5
            FOR row = 1 TO 10
                array_x(row,column)=(10*row + column)
                PRINT array_x(row,column);
            NEXT row
            PRINT
        NEXT column
        PRINT

        CALL forsub(array_x(,) BY REF)

END PROGRAM
```

Example 19–2 FORTRAN Subprogram

```
C       The FORTRAN subprogram receives
C       and then prints the same array

        SUBROUTINE forsub(f_array)
        INTEGER*2 f_array(5,10)
        DO 20 row = 1,5
            TYPE *, (f_array(row,column), column = 1,10)
20      CONTINUE
        RETURN
        END
```

You can pass only the data types that HP BASIC and FORTRAN have in common. You cannot pass a complex number from a FORTRAN program to a HP BASIC program, because HP BASIC does not support complex numbers. However, you can pass a complex number as two floating-point numbers and treat them independently in the HP BASIC program.

19.4 Calling System Routines

The steps for calling system routines are the same as those for calling any external routine. However, when calling system routines, you need to provide additional information, which is discussed in the following sections.

19.4.1 OpenVMS Run-Time Library Routines

The OpenVMS Run-Time Library routines are grouped according to the types of tasks they perform. The routines in each group have a prefix that identifies them as members of a particular OpenVMS Run-Time Library facility. Table 19–2 lists all the language-independent Run-Time Library facility prefixes and the types of tasks each facility performs.

Table 19–2 Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK$	DECtalk routines that are used to control the DECtalk device
LIB$	General purpose routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data
MTH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations
OTS$	Language-independent support routines that perform tasks such as data type conversions as part of a compiler's generated code
PPL$	Parallel processing routines that help you implement concurrent programs on single-CPU and multiprocessor systems
SMG$	Screen management routines that are used in designing, composing, and keeping track of complex images on a video screen
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings

19.4.2 System Service Routines

System services are system routines that perform a variety of tasks such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the OpenVMS Run-Time Library routines, which are divided into groups by facility, all system services share the same facility prefix (SYS$). However, these services are logically divided into groups that perform similar tasks. Table 19–3 describes these groups.

Table 19–3 System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of ASTs
Change Mode	Changes the access mode of particular routines
Condition Handling	Designates condition handlers for special purposes
Event Flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters
Information	Returns information about the system, queues, jobs, processes, locks, and devices
Input/Output	Performs I/O directly, without going through RMS
Lock Management	Enables processes to coordinate access to shareable system resources
Logical Names	Provides methods of accessing and maintaining pairs of character string logical names and equivalence names
Memory Management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data
Process Control	Creates, deletes, and controls execution of processes
Security	Enhances the security of OpenVMS systems
Time and Timing	Schedules events, and obtains and formats binary time values

19.4.3 System Routine Arguments

All of the system routine arguments are described in terms of the following information:

- OpenVMS usage
- Data type
- Type of access allowed
- Passing mechanism

OpenVMS usages are data structures that are layered on the standard OpenVMS data types. For example, the OpenVMS usage mask_longword signifies an unsigned longword integer that is used as a bit mask, and the OpenVMS usage floating_point represents any OpenVMS floating-point data type. Table 19–4 lists all the OpenVMS usages and the HP BASIC statements you need to implement them.

Table 19–4 OpenVMS Usages

OpenVMS Usage	BASIC Implementation
access_bit_names	Not applicable (NA)
access_mode	BYTE (signed)
address	LONG
address_range	LONG address_range[1] or RECORD address_range LONG beginning_address LONG ending_address END RECORD
arg_list	NA
ast_procedure	EXTERNAL LONG FUNCTION ast_proc [1]
boolean	LONG
byte_signed	BYTE
byte_unsigned	BYTE[2]
channel	WORD
char_string	STRING
complex_number	RECORD complex REAL real_part REAL imaginary_part END RECORD
cond_value	LONG
context	LONG
date_time	QUAD
device_name	STRING
ef_cluster_name	STRING
ef_number	LONG

[1]Use the LOC function to pass the address of an AST routine to a system service. Specify BY VALUE for the passing mechanism.

[2]Although unsigned data structures are not directly supported in BASIC, you can substitute the signed equivalent provided you do not exceed the range of the signed data structure.

(continued on next page)

Table 19–4 (Cont.) OpenVMS Usages

OpenVMS Usage	BASIC Implementation
exit_handler_block	RECORD EHCB LONG flink LONG handler_addr BYTE arg_count BYTE FILL(3) LONG status_value_addr END RECORD
fab	NA
file_protection	LONG
floating_point	SINGLE DOUBLE GFLOAT SFLOAT TFLOAT XFLOAT
function_code	RECORD function-code WORD major-function WORD subfunction END RECORD
identifier	LONG
io_status_block	RECORD iosb WORD iosb_field(1 to 4) END RECORD
item_list_2	RECORD item_list_two GROUP item(15) VARIANT CASE WORD comp_length WORD code LONG comp_address CASE LONG terminator END VARIANT END GROUP END RECORD

(continued on next page)

Table 19–4 (Cont.) OpenVMS Usages

OpenVMS Usage	BASIC Implementation
item_list_3	RECORD item_list_3 GROUP item (15) VARIANT CASE WORD buf_len WORD code LONG buffer_address LONG length_address CASE LONG terminator END VARIANT END GROUP END RECORD
item_list_pair	RECORD item_list_pair GROUP item(15) VARIANT CASE LONG code LONG item_value CASE LONG terminator END VARIANT END GROUP END RECORD item_list_pair
item_quota_list	RECORD item_quota_list GROUP quota(n) VARIANT CASE BYTE quota_name LONG item_value CASE BYTE list_end END VARIANT END GROUP END RECORD
lock_id	LONG
lock_status_block	NA
lock_value_block	NA
logical_name	STRING
longword_signed	LONG

(continued on next page)

Table 19–4 (Cont.) OpenVMS Usages

OpenVMS Usage	BASIC Implementation
longword_unsigned	LONG [2]
mask_byte	BYTE
mask_longword	LONG
mask_quadword	QUAD
mask_word	WORD
null_arg	A null argument is indicated by a comma used as a placekeeper in the argument list.
octaword_signed	BASIC$OCTAWORD[3]
octaword_unsigned	BASIC$OCTAWORD[3]
page_protection	LONG
procedure	EXTERNAL LONG FUNCTION proc
process_id	LONG
process_name	STRING
quadword_signed	QUAD
quadword_unsigned	QUAD [2]
rights_holder	QUAD
rights_id	LONG
rab	NA
section_id	QUAD
section_name	STRING
system_access_id	QUAD
time_name	STRING
uic	LONG
user_arg	LONG
varying_arg	Dependent upon application.
vector_byte_signed	BYTE array(n)
vector_byte_unsigned	BYTE array(n) [2]

[2]Although unsigned data structures are not directly supported in BASIC, you can substitute the signed equivalent provided you do not exceed the range of the signed data structure.

[3]The definition of the RECORD structures are included in the HP BASIC system definitions text library. See Section 19.4.4 for more information.

(continued on next page)

Table 19–4 (Cont.) OpenVMS Usages

OpenVMS Usage	BASIC Implementation
vector_longword_signed	LONG array(n)
vector_longword_unsigned	LONG array(n) [2]
vector_quadword_signed	QUAD array(n)
vector_quadword_unsigned	QUAD array(n)[2]
vector_word_signed	WORD array(n)
vector_word_unsigned	WORD array(n) [2]
word_signed	WORD
word_unsigned	WORD [2]

[2]Although unsigned data structures are not directly supported in BASIC, you can substitute the signed equivalent provided you do not exceed the range of the signed data structure.

If a system routine argument is optional, it will be indicated in the format section of the routine description in one of the following ways:

```
[,optional-argument]
,[optional-argument]
```

If the comma appears outside the brackets, you must either pass a zero by value or use a comma in the argument list as a placeholder to indicate the place of the omitted argument. If this is the last argument in the list, you must still include the comma as a placeholder. If the comma appears inside the brackets, you can omit the argument altogether as long as it is the last argument in the list.

19.4.4 Including Symbolic Definitions

To enhance program development, BASIC allows you to use symbolic definitions. **Symbolic definitions** are names or symbols associated with values. These symbols are used in many ways; the value associated with a symbol can be a status code, a mask, or an offset into a data structure. Many system routines depend on values that are defined in separate symbol definition files. For example, the status code for successful completion has a value of one; however, this code for successful completion is defined in the system library (STARLET) as the symbol SS$_NORMAL.

A program might compare the status code returned by a system service to either the symbolic constant SS$_NORMAL or the integer value one. The program would execute the same way in either case. In the first case, the

value for SS$_NORMAL is supplied at link time by the OpenVMS Linker. In the second case, the value 1 is included in the program as a literal constant.

The advantages of using symbolic definitions are as follows:

- Because symbolic definition names are mnemonic, the program is easier to read and understand.
- It is easier to write the symbolic definition and let the OpenVMS Linker fill in the value, than to look up the value associated with the symbol and include that value in the program.
- Should the value associated with a symbol ever change, you must relink the program. To change a hard-coded definition, you must edit the source file, then recompile and relink.

Symbolic definitions used by system services are located in the default system library, STARLET.OLB.

For Run-Time Library routines, the only time that you need to include symbolic definitions is when you are calling an SMG$ routine, or when you are calling a routine that is a jacket to a system service. (A jacket routine in the Run-Time Library is a routine that provides a simpler, more easily used interface to a system service.) If you call a routine in the SMG$ facility, you must include the definition file SMGDEF. All system services, however, require that you include SSDEF to check status. Many other system services require other symbol definitions as well.

To determine whether or not you need to include other symbolic definitions for the system service you want to reference, see the documentation for that service. If the documentation states that values are defined in the specified macro, you must include those symbolic definitions in your program. BASIC provides a text library that contains symbolic definitions that can be accessed using the %INCLUDE directive. In the following example, the definition file, SMGDEF is included from the text library SYS$LIBRARY:BASIC$STARLET.TLB:

```
%INCLUDE "SMGDEF" %FROM %LIBRARY "SYS$LIBRARY:BASIC$STARLET.TLB"
```

For more information about including text libraries, see Chapter 16.

19.4.5 Condition Values

Many system routines return a condition value that indicates success or failure. If a condition value is returned, you should check this value after you call a system routine and control returns to your program.

Condition values indicating success always appear first in the list of condition values for a particular routine, and success codes always have odd values. A success code that is common to many system routines is the condition value SS$_NORMAL, which indicates that the routine completed normally and successfully. You can test for this condition value as follows:

```
ret_status = SMG$CREATE_PASTEBOARD(pb_id)
IF (ret_status <> SS$_NORMAL) THEN
   CALL LIB$STOP(ret_status BY VALUE)
END IF
```

Because all success codes have odd values, you can check a return status for any success code. For example, you can cause execution to continue only if a success code is returned by including the following statements in your program:

```
ret_status = SMG$CREATE_PASTEBOARD(pb_id)
IF (ret_status AND 1%) = 0% THEN
   CALL LIB$STOP(ret_status BY VALUE)
END IF
```

In general, you can check for a particular success or failure code or you can test the condition value returned against all success codes or all failure codes.

19.5 Examples of Calling System Routines

This section provides complete examples of calling system routines from HP BASIC. In addition to the examples provided, the *VMS Run-Time Library Routines Volume* and the *HP OpenVMS System Services Reference Manual* also provide examples for selected routines. See these manuals for help about the use of a specific system routine.

Example 19–3 uses a function that invokes the SYS$TRNLNM system service. SYS$TRNLNM translates a logical name to an equivalence name. It places the equivalence name string into a string variable you supply in the parameter list.

System services never change a string variable's length. Therefore, if you use a system service that returns a string, be sure that the receiving string variable is long enough for the returned data. You can make sure of this in one of two ways:

- Define the string variable's length in a MAP, COMMON, or RECORD definition.

- Assign a long string to the variable (for example, A$ = SPACE$(80)). This pre-extends the variable so that it is long enough to receive all of the returned data.

Example 19–3 Calling System Services

```
10   !This function attempts to translate a logical name while searching
     !through all of the tables defined in LNM$DCL_LOGICAL.  If the translation
     !is successful, $TRNLNM returns the equivalence name string.

     FUNCTION STRING Translate(STRING Logical_name)
     EXTERNAL LONG FUNCTION SYS$TRNLNM (LONG, STRING, STRING, LONG, ITEM_LIST)
     EXTERNAL LONG CONSTANT LNM$M_CASE_BLIND, LNM$_STRING, SS$_NORMAL

     !Declare the parameters
     DECLARE LONG attributes,   &
                 trans_status
     DECLARE WORD equiv_len

     !Declare the value returned by the function.
     DECLARE LONG CONSTANT Buffer_length = 255
     RECORD item_list
     GROUP item (1)
             VARIANT
                  CASE
                     WORD Buf_len
                     WORD Code
                     LONG Buffer_address
                     LONG Length_address
                   CASE
                     LONG Terminator
              END VARIANT
      END GROUP item
      END RECORD item_list
      !Declare an instance of the record

      DECLARE ITEM_LIST TRNLNM_ITEMS

      !Define a common area for Translation_buffer

      COMMON (Trans_buffer) &
              STRING Translation_buffer = Buffer_length

      !Setting TRN$LNM to not distinguish between uppercase and lowercase
      !letters in the logical name to be translated.

      Attributes = LNM$M_CASE_BLIND
```

(continued on next page)

Example 19–3 (Cont.) Calling System Services

```
!Assign values to each record item.

TRNLNM_ITEMS::item(0)::Buf_len = Buffer_length
TRNLNM_ITEMS::item(0)::Code = LNM$_STRING
TRNLNM_ITEMS::item(0)::Buffer_address = LOC(Translation_buffer)
TRNLNM_ITEMS::item(0)::Length_address = LOC(Equiv_len)
TRNLNM_ITEMS::item(1)::Terminator = 0%

!Invoke the function

TRANS_STATUS = SYS$TRNLNM(attributes,"LNM$DCL_LOGICAL", logical_name, &
                          ,trnlnm_items)
!Check the condition value

IF trans_status AND SS$_NORMAL
THEN
        Translate = LEFT(Translation_buffer, Equiv_len)
ELSE
        Translate = ""
END IF
END FUNCTION
```

Example 19–4 is a program that demonstrates the use of the system service $QIOW. Unlike SYS$QIO, SYS$QIOW performs synchronously; SYS$QIOW returns a condition value to the caller after I/O operation is complete.

Example 19–4 Program Displaying the $QIOW System Service Routine

```
10  !Declare SYS$QIOW as an EXTERNAL FUNCTION

    EXTERNAL LONG FUNCTION SYS$QIOW(,WORD BY VALUE,LONG BY VALUE,WORD DIM() &
                                    BY REF,,,STRING BY REF,LONG BY VALUE,, &
                                    LONG BY VALUE,,)
    !Declare SYS$ASSIGN as an EXTERNAL FUNCTION

    EXTERNAL LONG FUNCTION SYS$ASSIGN(STRING,WORD,,)

    EXTERNAL LONG CONSTANT IO$_WRITEVBLK
```

(continued on next page)

Example 19–4 (Cont.) Program Displaying the $QIOW System Service Routine

```
!Declare the parameters

DECLARE STRING my_term, out_str, &
        WORD term_chan, counter, stat_block(3),&
        LONG ret_status, msg_len, car_cntrl

out_str = "Successful $QIOW output!"
my_term = "SYS$COMMAND"
msg_len = LEN(out_str)
car_cntrl = 32%
!Assign a channel to the terminal
ret_status = SYS$ASSIGN(my_term, term_chan, ,)
CALL LIB$STOP(ret_status BY VALUE) IF (ret_status AND 1%) = 0%
!Output the message four times
FOR counter = 1% to 4%
    ret_status = SYS$QIOW(,term_chan BY VALUE, IO$_WRITEVBLK BY VALUE, &
                          stat_block() BY REF,,,out_str BY REF,       &
                          msg_len BY VALUE,,car_cntrl BY VALUE,,)
    CALL LIB$STOP(ret_status BY VALUE) IF (ret_status AND 1%) = 0%
    CALL LIB$STOP(stat_block(0%) BY VALUE) &
                  IF (stat_block(0%) and 1%) = 0%
NEXT counter

END
```

Output

```
Successful $QIOW output!
Successful $QIOW output!
Successful $QIOW output!
Successful $QIOW output!
```

In addition to invoking the function SYS$QIOW, the previous example also invokes the function SYS$ASSIGN. This function provides a process with an I/O channel so that input and output operations can be performed on a logical device name *(my_term)*. As soon as SYS$ASSIGN is invoked and a path is established to the device, a counter is set up to invoke the $QIOW function four times. Once all I/O operations are complete, $QIOW returns to the caller.

19.6 OpenVMS Calling Standard

The primary purpose of the OpenVMS Calling Standard is to define the concepts for invoking routines and passing data between them. For more information, see the *HP OpenVMS Calling Standard*.

19.7 Additional Information

The information provided on system routines in this chapter is general to all system services and OpenVMS Run-Time Library routines. For specific information about these routines, see the *VMS Run-Time Library Routines Volume* and the *HP OpenVMS System Services Reference Manual*.

20

Libraries and Shareable Images

Libraries and shareable images allow you to access program symbols and incorporate commonly used routines into your source code. This chapter describes how to create and access libraries and shareable images in HP BASIC.

20.1 Overview of Libraries

Libraries are files that can contain object modules, text modules, and shareable images. There are two types of libraries: system-supplied and user-supplied. **System-supplied libraries** are provided by the OpenVMS system, and **user-supplied libraries** are libraries that you create.

Shareable images are similar to libraries; they contain code that can be shared by other programs. However, shareable images contain executable code rather than object code.

If you have routines that are used in many programs, placing the routines in object module libraries or shareable image libraries lets you access them at link time. You do not need to include the routines in the source code, thus shortening compilation time and conserving disk space.

If you have routines that are used simultaneously by many different programs, placing the routines in installed shareable images can improve performance at run time, conserve main physical memory, and reduce paging I/O because one copy of the executable code is shared by all users.

When you link programs, object module libraries, shareable image libraries, and shareable images can contain object code created by any native mode compiler or assembler.

For more understanding of libraries and shareable images, see the *HP OpenVMS Linker Utility Manual* and the *Guide to Creating OpenVMS Modular Procedures*. For more information about installing shareable images, see the *HP OpenVMS System Manager's Manual*. For information about text libraries, see Chapter 16.

20.2 System-Supplied Libraries

If symbols are unresolved after the OpenVMS Linker (linker) searches all user-supplied libraries, the linker goes on to search the files in the default system library. The OpenVMS system supplies the following libraries:

System Library	Description
IMAGELIB.OLB	Contains symbol tables for all Run-Time Library (RTL) shareable images that are part of the OpenVMS operating system—for example, an OpenVMS RTL routine is called Lib$FAO.
STARLET.OLB	An object module library containing the object files used to create the shareable image version of the OpenVMS RTL, and other less frequently used procedures. If program symbols remain unresolved after the OpenVMS Linker searches IMAGELIB.OLB, the linker then searches this library.

The linker searches modules in the following order:

1. Modules and libraries specified in the LINK command line, in the order given
2. User-supplied libraries (logicals of the form LNK$LIBRARY and LNK$LIBRARY_1 through LNK$LIBRARY_999)
3. Images contained in IMAGELIB.OLB
4. Modules contained in STARLET.OLB

The linker only includes references to needed shareable images in the image being created. You can use the /NOSYSSHR qualifier to the LINK command to suppress the linker's search of RTL shareable images. Similarly, you can use the /NOSYSLIB qualifier to suppress the linker's search of both RTL shareable images and STARLET.OLB.

The linker searches user-supplied libraries before searching the default system library. If one of your modules has the same name (program symbol) as an OpenVMS System Service or an RTL routine, the linker includes your module in the resulting image rather than the system service or RTL routine.

20.3 Creating User-Supplied Object Module Libraries

You create a user-supplied object module library with the DCL command LIBRARY. Specify a library file specification as well as a list of the program modules you want to insert into the library. For example:

```
$ BASIC MODULE1,MODULE2
$ LIBRARY/CREATE TESTLIB1.OLB MODULE1.OBJ,MODULE2.OBJ
```

In the previous example, the BASIC command creates object files from MODULE1.BAS and MODULE2.BAS. The LIBRARY command creates an object module library named TESTLIB1.OLB and inserts MODULE1.OBJ and MODULE2.OBJ into that library. See the *HP OpenVMS DCL Dictionary* for more information about the LIBRARY command.

20.3.1 Accessing User-Supplied Object Module Libraries

To access user-supplied object module libraries, specify the /LIBRARY qualifier to the DCL command LINK. For example:

```
$ LINK MAIN,TESTLIB/LIBRARY
```

This command instructs the linker to search the library TESTLIB.OLB for any unresolved symbols in the HP BASIC object module MAIN.OBJ.

Also, you can explicitly include a module from a library with the /INCLUDE qualifier. For example:

```
$ LINK MAIN,TESTLIB/LIBRARY/INCLUDE = (module1,module2)
```

This command instructs the linker to include *module1* and *module2* from the library TESTLIB.OLB, whether or not it needs these modules to resolve symbols.

You can access user-supplied object module libraries automatically. However, a program executing at DCL level does not automatically search libraries that are assigned to the logical name BASIC$LIB0. Instead, the linker searches libraries that are assigned to the logical name LNK$LIBRARY. If you have more than one library for the linker to search, you must number these libraries consecutively; otherwise, the linker does not search past the first missing logical name. The linker allows you to number libraries from 1 to 999.

For example:

```
$ DEFINE LNK$LIBRARY USER$$DEV:[KELLY]TESTLIB.OLB
$ DEFINE LNK$LIBRARY_1 USER$$DEV:[KELLY]TESTLIB1.OLB
$ DEFINE LNK$LIBRARY_2 USER$$DEV:[KELLY]TESTLIB2.OLB
```

After you issue these commands, a program executing at DCL level automatically accesses these three library files to resolve program symbols.

20.4 Shareable Images

Shareable images are not directly executable. They contain executable code that can be shared by other images and are intended to be included by the linker in other images.

The benefits of using shareable images include:

- Conserving disk storage space
- Conserving main physical memory
- Reducing paging I/O
- Allowing shared memory-resident databases
- Eliminating the need to relink programs that access a new version of a shared routine

Note

Some of these benefits can only be realized if the shareable image is installed with the OpenVMS Install utility (Install).

To create a shareable image, use the /SHAREABLE qualifier with the DCL command LINK and specify at least one object module. For example:

```
$ LINK/SHAREABLE prog1
```

This command creates an image that can be linked to other programs. You cannot execute a shareable image with the DCL command RUN.

When a program is linked with a shareable image, the required shareable image code is not included in the created executable image on the disk. This code is included by the image activator at run time. Therefore, many programs can reside on disk and be bound with a particular shareable image, and only one physical copy of that shareable image file needs to exist on disk.

If a shareable image has been installed using the OpenVMS Install utility, you conserve physical memory and potentially reduce paging I/O. Many processes can include the physical memory pages of an installed shareable image in their address space. This reduces the requirements for physical memory.

Paging occurs when a process attempts to access a virtual address that is not in the process working set. When this page fault occurs, the page is either in a disk file, in which case paging I/O is required, or is already in physical memory. If a page fault occurs for a shared page, the shared page may already be resident in memory and no paging I/O is required.

20.4.1 Accessing Shareable Images

To access a shareable image, follow these steps:

1. Write and compile a program unit that is to be inserted into a shareable image.
2. Create an options file required for the link operation.
3. Link the program with the /SHAREABLE qualifier, and specify the options file with the /OPTION qualifier.
4. Write a main program that accesses the routine in the shareable image.
5. Compile the main program, and link it with the shareable image.

The following example shows how to access a shareable image by performing these steps:

1. Write and compile a program unit that is to be inserted into a shareable image.

```
!Program name - ADD.BAS
FUNCTION REAL ADD (LONG A, LONG B)
ADD = A + B
FUNCTIONEND
```

2. Create an options file that will export the function for the link operation.

```
! Program name - ADDSUB.OPT
SYMBOL_ADDER = (ADD=PROCEDURE)
```

3. Link the program with the qualifiers /SHAREABLE and /OPTION.

```
$ LINK/SHAREABLE ADD, ADDSUB/OPTION
```

   Copy the shareable image to SYS$SHARE:, or define a logical name to the full image file specification. For example,

```
$ Define ADD Sys$Login:Add.exe
```

4. Write a main program that accesses the routine in the shareable image.

```
!Program name - CALLADD.BAS
EXTERNAL REAL FUNCTION ADD (LONG, LONG)
DECLARE LONG X,Y
X = 1
Y = 2
PRINT ADD(X,Y)
END
```

5. Compile the main program, and link it with the shareable image.

```
$ LINK CALLADD,ADDMAIN/OPTION
```

To link CALLADD with the shareable image ADD, you must have a linker options file specifying that ADD is a shareable image. For example:

```
!Options file - ADDMAIN.OPT
ADD/SHAREABLE
```

Next, execute the program. Upon executing the program, the image activator attempts to locate the shareable image in the directory SYS$SHARE:. If you want the image activator to access a shareable image outside SYS$SHARE:, you must define a logical name to the shareable image before you execute the program. Define the full file specification of the shareable image to the name of the shareable image, as follows:

```
$ DEFINE MYSHR DISK$WORKDISK:[MYDIR]MYSHR.EXE
```

This is a simple example of using shareable images. For more information, see the *HP OpenVMS Linker Utility Manual* and the *Guide to Creating OpenVMS Modular Procedures*.

21

Using CDD/Repository with BASIC

This chapter explains how you can take advantage of CDD/Repository capabilities. For more detailed information about CDD/Repository, see *Using CDD/Repository on VMS Systems*.

21.1 Overview of CDD/Repository

CDD/Repository is a common data dictionary tool that supports sharing of data definitions by OpenVMS programming languages and information architecture products. Each language or product translates the generic definitions stored in CDD/Repository language- or product-specific definitions that it can use.

BASIC supports CDD/Repository features including dependency recording. Dependency recording allows you to record (or track) which programs use CDD/Repository data definitions. Dependency recording helps evaluate the effort needed to change a record definition by identifying the modules that need to be modified, recompiled, or both.

To support dependency recording, CDD/Repository uses a dictionary structure known as CDO-format. (The type of dictionary used in CDD versions prior to Version 4.0 is known as DMU-format.) You can have many CDO-format dictionaries on an OpenVMS system (but only one DMU-format dictionary). The two types of dictionaries can coexist on a system, and a program can refer to data definitions in both types.

21.2 CDD/Repository Concepts

This section introduces CDD/Repository concepts.

21.2.1 Dictionary Formats

CDD/Repository allows the following types of dictionaries:

- DMU-format dictionary
- CDO-format dictionary

These dictionaries can coexist on a system to form one logical directory structure. CDD/Repository uses a special dictionary, known as the compatibility dictionary, that allows an application to refer to dictionary definitions without concern about which type of dictionary format the definitions are stored in.

The compatibility dictionary is a CDO-format dictionary whose directory hierarchy matches that of the DMU-format dictionary (if any) on the system.

Note

The compatibility dictionary is an installation option for CDD/Repository. If there is no compatibility dictionary, an application program can refer to both types of dictionaries. In this case, refer to the CDO-format dictionary with an anchor origin path name and to the DMU-format dictionary with a CDD$TOP path name. Anchor origin path names are described in Section 21.2.2.

Refer to the CDD/Repository documentation for detailed information about the CDO utility and the compatibility dictionary.

21.2.2 Dictionary Path Names

To access dictionary definitions, you must specify a path name in the %INCLUDE %FROM %CDD or %REPORT %DEPENDENCY directive. The path name tells CDD/Repository where to locate a particular data definition in its directory. A CDD/Repository path name consists of a string of names separated by periods and enclosed in quotation marks.

The origin is the top, or root, of a dictionary directory. This directory contains other dictionary directories, subdictionary directories, and objects.

HP BASIC allows the following types of valid path name parameters when referring to CDO dictionary definitions. They differ in the method of specifying the dictionary origin.

- Dictionary anchor path name

An anchor path name begins with an anchor, which is an OpenVMS directory specification, as the dictionary origin. The anchor specifies the OpenVMS directory that contains the CDO dictionary. This is known as the CDO naming convention. In the following example, MYNODE::DISK$2:[MYDIRECTORY] is the anchor:

```
MYNODE::DISK$2:[MYDIRECTORY]PERSONNEL.EMPLOYEES_REC
```

- CDD$TOP path name

  Use this to refer to either DMU-format dictionary definitions or CDO-format dictionary definitions in a compatibility dictionary. The path origin is always CDD$TOP. This is known as the DMU naming convention. For example:

  ```
  CDD$TOP.PERSONNEL.EMPLOYEES_REC
  ```

- Relative path name

  CDD/Repository always begins its search at CDD$TOP (or at the anchor you specify) unless you define another directory or object to be the start of your directory. You can do this by assigning the name of a dictionary directory to the logical name CDD$DEFAULT. For example:

  ```
  $ DEFINE CDD$DEFAULT CDD$TOP.BASIC
  ```

  Using this command defines the dictionary directory CDD$TOP.BASIC as the default start of your directory. You can override the defined default by specifying CDD$TOP in a path name.

  You can omit the origin of a path name and specify a relative path name. Any path name that does not begin with either CDD$TOP or an anchor is automatically appended to the current CDD$DEFAULT. For example, you can specify:

  ```
  PERSONNEL.EMPLOYEES_REC
  ```

  If CDD$DEFAULT is MYNODE::MY$DISK:[MYDIR], the relative path name is the same as:

  ```
  MYNODE::MY$DISK:[MYDIR]PERSONNEL.EMPLOYEES_REC.
  ```

  Similarly, if CDD$DEFAULT is CDD$TOP.MYDIR, the relative path name is the same as:

  ```
  CDD$TOP.MYDIR.PERSONNEL.EMPLOYEES_REC.
  ```

21.2.3 Dictionary Entities

Several types of entities can exist in a dictionary. For example, DMU-format and CDO-format dictionaries each contain record entities, database entities, and form entities.

HP BASIC creates a compiled module entity (and relationships in CDD/Repository dictionaries that depend on compiled module entities) only if the compilation generates an object file. Therefore, compiled module entities are not generated if you specify the /NOOBJECT qualifier on the command line or if the program has compilation errors.

21.2.4 Dictionary Relationships

Relationships occur in a CDO-format dictionary when two or more CDO entity definitions are connected in any of several possible ways. For example, you can relate a set of field definitions to a record definition by including the field names in the record definition.

See the CDD/Repository documentation for detailed information about relationships in a CDO-format dictionary.

21.2.5 Extracting CDD/Repository Data Definitions

A data definition is one type of a CDD/Repository object. In HP BASIC, you can extract only data definition objects into your program.

To extract a CDD/Repository data definition in HP BASIC, specify the %INCLUDE %FROM %CDD compiler directive and a CDD/Repository path name. You can use this to extract a data definition from either a DMU-format or CDO-format dictionary. For example:

```
%INCLUDE %FROM %CDD "CDD$TOP.BASIC.BASICDEF"
```

The %INCLUDE %FROM %CDD directive extracts the CDD/Repository data definition you specify and translates it into HP BASIC syntax. In HP BASIC, the syntax for data definitions or data structures is defined by the RECORD statement.

After a CDD/Repository data definition is translated into RECORD statement syntax, you can reference the name of the RECORD statement in your HP BASIC programs. After compilation, the translated RECORD statement is included as a part of your program's listing.

The following is an example of a CDD/Repository data definition and the translated HP BASIC RECORD statement. The examples in this chapter are CDD/Repository data definitions in DMU-format that were written in the Common Data Definition Language (CDDL). In all examples, a CDDL

data definition is displayed in lowercase letters, and the translated RECORD statement is displayed in uppercase letters.

CDDL Definition

```
define record
CDD$top.basic.basicdef
        description is

           /* This is an example record containing
           only data types supported by HP BASIC */.

        employee structure.
                street          datatype is text
                                size is 30 characters.
                city            datatype is text
                                size is 30 characters.
                state           datatype is text
                                size is 2 characters.

                zip_code structure.
                    new         datatype is packed decimal
                                size is 4 digits.
                    old         datatype is packed decimal
                                size is 5 digits.
                end zip_code structure.

                emp_number      datatype is signed word.
                wage_class      datatype is text
                                size is 2 characters.
                salary_ytd      datatype is d_floating.
        end employee structure.
end basicdef.
```

Translated RECORD Statement

```
        C1              !   This is an example record containing
        C1              !   only data types supported by HP BASIC
        C1              RECORD  EMPLOYEE                    ! UNSPECIFIED
        C1                STRING  STREET  = 30              ! TEXT
        C1                STRING  CITY  = 30                ! TEXT
        C1                STRING  STATE  = 2                ! TEXT
        C1                GROUP   ZIP_CODE                  ! UNSPECIFIED
        C1                  DECIMAL(4 ,0 ) NEW              ! PACKED DECIMAL
        C1                  DECIMAL(5 ,0 ) OLD              ! PACKED DECIMAL
        C1                END GROUP
        C1                WORD    EMP_NUMBER                ! SIGNED WORD
        C1                STRING  WAGE_CLASS  = 2           ! TEXT
        C1                DOUBLE  SALARY_YTD                ! D_FLOATING
        C1              END RECORD
```

When HP BASIC translates a CDD/Repository data definition, it does the following:

- For DMU-format definitions, BASIC takes the field name specified in the first CDDL STRUCTURE statement and assigns that name to HP BASIC RECORD. For CDO-format definitions, BASIC takes the record name from the CDO DEFINE RECORD statement and assigns that name to a HP BASIC RECORD. In the previous example, the first CDD/Repository structure statement is *employee structure*. When HP BASIC translates this line of a CDD/Repository data definition, it names the record *EMPLOYEE*. If this first structure is unnamed, HP BASIC signals the error "Record from CDD/Repository does not have a record name."
- Translates the field name in any subsequent CDD/Repository STRUCTURE statement to be the name of a group. In the previous example, the second STRUCTURE statement, *zip_code structure*, is translated to *GROUP ZIP_ CODE*.
- Translates subordinate field names in CDD/Repository STRUCTURE statements to elementary components in the RECORD statement. In the previous example, the subordinate field name *street* is translated to *STRING STREET*.

If you specify the /LIST qualifier when HP BASIC translates a CDD/Repository data definition, it does the following:

- Begins each line of the RECORD statement with the letter "C" followed by a number. The letter "C" tells you that the RECORD statement is translated from a CDD/Repository data definition. The number tells you the nesting level of the %INCLUDE %FROM %CDD directive within the source program. For example, if your source program directly extracts a CDD/Repository record definition, then each line is preceded by "C1." If the CDD/Repository extraction came from a file included in the source program, then each line of the record definition is preceded by "C2," and so on.
- Includes the explanatory text in the CDDL DESCRIPTION clause as comment fields.
- Translates the data type text in the subordinate field to a comment field that tells you the data type of each elementary RECORD component. For example, the comment *! TEXT* tells you that *STRING STREET* is a text data type.

HP BASIC requires that a CDD/Repository data definition include a minimum of one structure to be translated into a RECORD statement. If a CDD/Repository data definition contains only a single subordinate field (without a structure), HP BASIC signals an error because it cannot give a name to the RECORD statement. You cannot include a CDO FIELD definition in a HP BASIC program. You can, however, include CDO RECORD definitions that contain that field.

For more information about how HP BASIC translates CDD/Repository data types, see Section 21.9.

21.3 Using CDD/Repository with BASIC

When dependency recording is in effect, the compiler updates the CDO-format dictionary to show what dictionary data entities are used by the program (the data dependencies created by the compilation).

To take advantage of dependency recording, do the following:

- Use either or both of the HP BASIC lexical directives, %INCLUDE %FROM %CDD and %REPORT %DEPENDENCY, in the source program to define the dependency relationships you want to create between your program and definitions in the CDO-format dictionary.
- Establish a CDO-format dictionary called CDD$DEFAULT.
- Include the /DEPENDENCY_DATA qualifier in the BASIC command that compiles the module.

21.3.1 /DEPENDENCY_DATA Qualifier

When you compile a program that references CDO-format data definitions, you can include the qualifier in the BASIC command line. The /DEPENDENCY_DATA qualifier instructs the compiler to create dependency relationships (as defined in the program by %INCLUDE and %REPORT directives) and update the dictionary to show those relationships.

To prevent update of the dictionary, specify the default, /NODEPENDENCY_DATA. The compiler can extract record definitions from the dictionary (as specified by %INCLUDE %FROM %CDD directives in the program) but not update the dictionary. The compilation does not add compiled module entities and file entities to the dictionary, nor does it create dependency relationships in the dictionary, unless you specify the /DEPENDENCY_DATA qualifier.

21.3.2 Creating Relationships with Included Record Definitions

In Section 21.2.4 a record description is defined as a set of fields (thus establishing a simple relationship in CDD/Repository between the record and its fields). With that record description defined, you can include it in a HP BASIC program.

With either a DMU-format or CDO-format dictionary, the compiler can extract a record description into a program. Use the %INCLUDE lexical directive in the source program. The format is as follows:

```
%INCLUDE %FROM %CDD "pathname"
```

For example, the following BASIC source code extracts a record description named *ADDRESS_REC* from CDD/Repository:

```
PROGRAM EXAMPLE1
%INCLUDE %FROM %CDD "CDD$TOP.SMITH.ADDRESS_REC"
DECLARE ADDRESS_REC TEST_RECORD
INPUT "First name";TEST_RECORD::FIRST_NAME
INPUT "Last name";TEST_RECORD::LAST_NAME
INPUT "Address";TEST_RECORD::ADDRESS
INPUT "City";TEST_RECORD::CITY
INPUT "State";TEST_RECORD::STATE
INPUT "Zip code";TEST_RECORD::ZIP_CODE

PRINT TEST_RECORD::FIRST_NAME; TEST_RECORD::LAST_NAME
PRINT TEST_RECORD::ADDRESS
PRINT TEST_RECORD::CITY; TEST_RECORD::STATE; TEST_RECORD::ZIP_CODE
```

The following shows the content of the record:

```
 1            PROGRAM EXAMPLE1
 2            %INCLUDE %FROM %CDD "CDD$TOP.SMITH.ADDRESS_REC"
   C1              RECORD  ADDRESS_REC               ! UNSPECIFIED
   C1                STRING  FIRST_NAME  = 20        ! TEXT
   C1                STRING  LAST_NAME  = 30         ! TEXT
   C1                STRING  ADDRESS  = 40           ! TEXT
   C1                STRING  CITY  = 20              ! TEXT
   C1                STRING  STATE  = 2              ! TEXT
   C1                DECIMAL(5 ,0 ) ZIP_CODE         ! PACKED DECIMAL
   C1              END RECORD
 3            DECLARE ADDRESS_REC TEST_RECORD
 4            INPUT "First name";TEST_RECORD::FIRST_NAME
 5            INPUT "Last name";TEST_RECORD::LAST_NAME
 6            INPUT "Address";TEST_RECORD::ADDRESS
 7            INPUT "City";TEST_RECORD::CITY
 8            INPUT "State";TEST_RECORD::STATE
 9            INPUT "Zip code";TEST_RECORD::ZIP_CODE
10
```

```
11            PRINT TEST_RECORD::FIRST_NAME; TEST_RECORD::LAST_NAME
12            PRINT TEST_RECORD::ADDRESS
13            PRINT TEST_RECORD::CITY; TEST_RECORD::STATE; TEST_RECORD::ZIP_CODE
```

With a CDO-format dictionary, you can also instruct the dictionary to create and maintain a formal relationship between the record description and the compiled module entity that represents your program in the dictionary.

This is known as a CDD$COMPILED_DEPENDS_ON relationship. Specify the /DEPENDENCY_DATA qualifier when compiling a program as follows:

```
$ BASIC/DEPENDENCY_DATA EX1.BAS
```

If you specify the /DEPENDENCY_DATA qualifier, the compiled module entity is created and updated to reflect that your program uses that record. If you want to change the data definition, CDO allows you to find out what programs depend on it before doing so. For example:

```
CDO> DIRECTORY
 Directory SYS$COMMON:[CDDPLUS]SMITH
ADDRESS;1                                       FIELD
ADDRESS_REC;1                                   RECORD
CITY;1                                          FIELD
EXAMPLE1;1                                      CDD$COMPILED_MODULE
FIRST_NAME;1                                    FIELD
LAST_NAME;1                                     FIELD
STATE;1                                         FIELD
ZIP_CODE;1                                      FIELD
   .
   .
   .
```

You can use the CDO SHOW USES command to find out what programs use a dictionary definition. For example:

```
CDO> SHOW USES ADDRESS_REC
Owners of SYS$COMMON:[CDDPLUS]SMITH.ADDRESS_REC;1
|   SYS$COMMON:[CDDPLUS]SMITH.EXAMPLE1;1  (Type : CDD$COMPILED_MODULE)
|   |   via CDD$COMPILED_DEPENDS_ON
```

You can also use CDO to find out what dictionary definitions a program uses. For example:

```
CDO> SHOW USED_BY EXAMPLE1
Members of SYS$COMMON:[CDDPLUS]SMITH.EXAMPLE1;1
|   EX1                             (Type : CDD$FILE)
|   |   via CDD$IN_FILE
|   SYS$COMMON:[CDDPLUS]SMITH.ADDRESS_REC;1  (Type : RECORD)
|   |   via CDD$COMPILED_DEPENDS_ON
```

21.4 Creating Relationships for Referenced Dictionary Entities

The compiler can create a relationship between a compiled module entity and any dictionary entity that a program references (such as a VAX Rdb/VMS database or a form definition). The referenced dictionary entity is not copied to the program. Instead, the compiled program references the dictionary entity at run time or with the help of a preprocessor.

To create relationships for referenced dictionary entities in a BASIC program, use the %REPORT %DEPENDENCY lexical directive in the source program and specify the /DEPENDENCY_DATA qualifier when you compile the program. The format is as follows:

%REPORT %DEPENDENCY *"pathname" ["relationship-type"]*

The "pathname" parameter identifies the dictionary item that the compiled object module references. The path name can specify a CDO-format dictionary item (with an anchor as the first element), or it can specify a CDO-format item in the compatibility dictionary (which can be specified either as a CDD$TOP path name or as an anchor path name). See Section 21.2.2 for a full description of the path name options.

The optional “relationship-type” parameter determines the type of relationship by specifying a CDD/Repository protocol. There are many valid values; refer to the CDD/Repository documentation for full information. The most commonly used relationship for HP BASIC users is as follows:

```
CDD$COMPILED_DEPENDS_ON
```

This specifies a relationship that links a compiled object module to the element that goes into the compilation. This is the default.

The %REPORT %DEPENDENCY directive is meaningful only when the following conditions are true:

- The /DEPENDENCY_DATA qualifier is specified in the BASIC command line. (If it is not specified, the compiler checks syntax but does not update the dictionary to reflect this usage of the item.)
- The current CDD$DEFAULT dictionary points to a directory in a CDO dictionary.
- The dictionary item specified by *pathname* is in a CDO-format dictionary. (No relationship can be created in a DMU-format dictionary.)

Suppose the HP BASIC program DOG_REPORT.BAS contains the following directive:

```
%REPORT %DEPENDENCY "DISK1$:[CDDPLUS.BASIC]SMITH.DOG_DATABASE"
```

Use the /DEPENDENCY_DATA qualifier when you compile the program:

```
$ BASIC/DEPENDENCY_DATA DOG_REPORT
```

After the compilation, the dictionary contains the following:

```
CDO> DIR

 Directory DISK1$:[CDDPLUS.BASIC]SMITH

BREED;1                                          FIELD
CALL_NAME;1                                      FIELD
DOG_REPORT$MAIN;1                                CDD$COMPILED_MODULE
DOG_DATABASE;1                                   CDD$DATABASE
DOG_INFORMATION;1                                CDD$RMS_DATABASE
DOG_REC;1                                        RECORD
OWNER_NUMBER;1                                   FIELD
REG_DOG_NAME;1                                   FIELD

CDO> SHOW USES DOG_DATABASE

Owners of DISK1$:[CDDPLUS.BASIC]SMITH.DOG_DATABASE;1
|   DISK1$:[CDDPLUS.BASIC]SMITH.DOG_REPORT$MAIN;1  (Type : CDD$COMPILED_MODULE)
|   |   via CDD$COMPILED_DEPENDS_ON

CDO> SHOW USED_BY DOG_REPORT$MAIN
Members of DISK1$:[CDDPLUS.BASIC]SMITH.DOG_REPORT$MAIN;1
|   DOG_REPORT                      (Type : CDD$FILE)
|   |   via CDD$IN_FILE
|   DISK1$:[CDDPLUS.BASIC]SMITH.DOG_DATABASE;1  (Type : CDD$DATABASE)
|   |   via CDD$COMPILED_DEPENDS_ON
```

21.5 Specifying a CDD History List Entry

When your HP BASIC program accesses CDD/Repository, you have the option of entering a history list entry in the database. The history list entry provides a history of users that access CDD/Repository.

You create a history list entry by specifying the DCL command qualifier /AUDIT. For example:

```
$ BASIC/DEPENDENCY_DATA/AUDIT="History text goes here" EX1.BAS
```

Note that instead of typing the text directly on the command line, you can also specify a file specification that contains the history entry.

When you specify /AUDIT, a history list entry is created for each compiled module entity that the compilation creates. In addition, the compilation will add a history list entry to each data definition that your program extracts with the %INCLUDE %FROM %CDD directive.

You can display history list information using the CDO utility. For example:

```
CDO> SHOW GENERIC CDD$COMPILED_MODULE EXAMPLE1 /AUDIT
Definition of EXAMPLE1   (Type : CDD$COMPILED_MODULE)
|   History entered by SMITH ([SMITH])
|       using BASIC Vn.n
|       to CREATE definition on 25-APR-1989 13:04:01.48
|       Explanation:
|               "History text goes here"
```

21.6 CDD/Repository Arrays

CDD/Repository supports the following arrays:

- Multidimensional arrays (the ARRAY clause)
- One-dimensional, fixed length arrays (the OCCURS clause or ARRAY clause)
- One-dimensional, variable length arrays (the OCCURS DEPENDING ON clause—note that HP BASIC does not support this clause)

Arrays are valid for any CDD/Repository field. HP BASIC does not support dimensions on a RECORD statement; therefore, you cannot declare an entire RECORD statement as an array. However, you can dimension an instance of the record.

The following is an example of a CDDL data definition containing arrays and the corresponding HP BASIC RECORD statement:

CDDL Definition

```
define record CDD$top.basic.array1
    description is

        /* test arrays */.

    array_1 structure.
        my_byte         array 0:2           datatype        signed byte.
        my_string       array 0:10          datatype        text size 10.
        my_s_real       array 0:2 0:4       datatype        f_floating.
        my_d_real       array 1:3           datatype        d_floating.
        my_g_real       occurs 4 times      datatype        g_floating.
        my_h_real       occurs 4 times      datatype        h_floating.
    end array_1 structure.
end array1.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.ARRAY1"
        C1              !   test arrays
        C1              RECORD  ARRAY_1                     ! UNSPECIFIED
        C1                BYTE    MY_BYTE(0 TO 2)           ! SIGNED BYTE
        C1                STRING  MY_STRING(0 TO 10) = 10   ! TEXT
        C1                SINGLE  MY_S_REAL(0 TO 2,0 TO 4) ! F_FLOATING
        C1                DOUBLE  MY_D_REAL(1 TO 3)         ! D_FLOATING
        C1                GFLOAT  MY_G_REAL(1 TO 4)         ! G_FLOATING
        C1                HFLOAT  MY_H_REAL(1 TO 4)         ! H_FLOATING
        C1              END RECORD
```

By default, arrays in CDD/Repository are row-major. This means that when storage is allocated for the array, the rightmost subscript varies fastest. All HP BASIC arrays are row-major. HP BASIC does not support column-major arrays. If a CDD/Repository definition containing a column-major array is extracted, HP BASIC signals the error "<array-name> from CDD/Repository is a column major array."

By default, HP BASIC extracts an array field from CDD/Repository with the bounds specified in the data definition. However, if you use the qualifier /OLD_VERSION[=CDD_ARRAYS] when you extract a data definition, HP BASIC translates the data definition with lower bounds as zero and adjusts the upper bounds. This means that an array with dimensions of (2,5) in CDD/Repository is translated by HP BASIC to be an array with a lower bound of 0 and an upper bound of 3. HP BASIC issues an informational message to confirm the array bounds when you use this qualifier.

The following CDD/Repository data definition and corresponding RECORD statement are extracted with the /OLD_VERSION[=CDD_ARRAYS] qualifier:

CDDL Definition

```
define record CDD$top.basic.array2
    description is

        /* test arrays with /old_version[=CDD_ARRAYS] qualifier */.

    array_2 structure.
        my_byte         array 0:2           datatype        signed byte.
        my_string       array 0:10          datatype        text size 10.
        my_s_real       array 0:2 0:4       datatype        f_floating.
        my_d_real       array 1:3           datatype        d_floating.
        my_g_real       occurs 4 times      datatype        g_floating.
        dep_item                            datatype        signed longword.
        my_h_real       occurs 4 times
                                            datatype        h_floating.
    end array_2 structure.
end array2.
```

Translated RECORD Statement

```
1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.ARRAY2"
   C1              !   test arrays with /old_version[=CDD_ARRAYS] qualifier
   C1              RECORD  ARRAY_2                    ! UNSPECIFIED
   C1                BYTE    MY_BYTE(0 TO 2)          ! SIGNED BYTE
   C1                STRING  MY_STRING(0 TO 10) = 10  ! TEXT
   C1                SINGLE  MY_S_REAL(0 TO 2,0 TO 4) ! F_FLOATING
   C1                DOUBLE  MY_D_REAL(0 TO 2)        ! D_FLOATING
   C1                GFLOAT  MY_G_REAL(0 TO 3)        ! G_FLOATING
   C1                LONG    DEP_ITEM                 ! SIGNED LONGWORD
   C1                HFLOAT  MY_H_REAL(0 TO 3)        ! H_FLOATING
   C1              END RECORD
```

21.7 CDD/Repository Variants

A variant comprises two or more fields of a record that provide alternative descriptions for the same portion of a record.

The following is an example of a CDDL data definition containing variant fields and the corresponding HP BASIC RECORD statement:

CDDL Definition

```
define record CDD$top.basic.variant_example
    description is

        /* test simple variant */.
    variant_example structure.
        my_string datatype text size 9.
        variants.
            variant.
                my_s_real       datatype        f_floating.
                my_d_real       datatype        d_floating.
            end variant.
            variant.
                my_g_real       datatype        g_floating.
                my_h_real       datatype        h_floating.
            end variant.
        end variants.
                my_byte         datatype        signed byte.
   end variant_example structure.
end variant_example.
```

Translated RECORD Statement

```
1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.VARIANT_EXAMPLE"
   C1              !   test simple variant
   C1              RECORD  VARIANT_EXAMPLE            ! UNSPECIFIED
   C1                STRING  MY_STRING  = 9           ! TEXT
   C1                VARIANT
   C1                CASE
   C1                  SINGLE  MY_S_REAL              ! F_FLOATING
   C1                  DOUBLE  MY_D_REAL              ! D_FLOATING
   C1                CASE
   C1                  GFLOAT  MY_G_REAL              ! G_FLOATING
   C1                  HFLOAT  MY_H_REAL              ! H_FLOATING
   C1                END VARIANT
   C1                BYTE    MY_BYTE                  ! SIGNED BYTE
   C1              END RECORD
```

CDD/Repository data definitions sometimes contain VARIANTS OF field description statements as well as simple variants. A CDDL or CDO VARIANTS OF statement names a tag variable whose value at run time determines which of the variant fields is the current variant. HP BASIC does not support the VARIANTS OF statement. If a CDD/Repository data definition containing a VARIANTS OF statement is extracted, HP BASIC signals the informational message, "<number> tag value from CDD/Repository ignored" and treats the VARIANTS OF as an ordinary variant and ignores the tag value.

21.8 NAME FOR BASIC Clause

HP BASIC supports the CDDL and CDO field attribute clause NAME FOR BASIC.

The field attribute clause NAME FOR BASIC declares a facility-specific name for a field. For example:

```
name for basic is "subject_name$"
```

When you assign a name using the NAME FOR BASIC clause in a CDDL or CDO data definition, HP BASIC recognizes only this name when you refer to the field. Note that when you use the NAME FOR BASIC clause, you can place dollar sign ($) and percent sign (%) suffixes in your RECORD statement field names.

The following example is a CDDL data definition containing the NAME FOR BASIC clause and the corresponding HP BASIC RECORD statement.

CDDL Definition

```
define record city_study
    description is

       /* This example formats data resulting from a
       study on the relationship between place of birth
       and earning potential */.
    info structure.
        subject_name                    datatype text size 10
                                        name for basic is "subject_name$".
        birth_city                      datatype text size 10
                                        name for basic is "city_of_birth$".
        salary                          datatype signed byte
                                        name for basic is "salary%".
    end info structure.
end city_study.
```

Translated RECORD Statement

```
1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.CITY_STUDY"
   C1              !   This example formats data resulting from a
   C1              !   study on the relationship between place of birth
   C1              !   and earning potential
   C1              RECORD  INFO                        ! UNSPECIFIED
   C1                STRING  SUBJECT_NAME$  = 10       ! TEXT
   C1                STRING  CITY_OF_BIRTH$  = 10      ! TEXT
   C1                BYTE    SALARY%                   ! SIGNED BYTE
   C1              END RECORD
```

Caution

The NAME FOR BASIC clause enables you to assign completely different names to the same field.

For more information about the CDDL NAME FOR BASIC field attribute clause, see the CDD/Repository documentation.

21.9 CDD/Repository Data Types

HP BASIC supports a subset of CDD/Repository data types, as shown in Table 21–1.

Table 21–1 Supported CDD/Repository Data Types

Data Type	HP BASIC Translation
TEXT	STRING
SIGNED BYTE	BYTE
SIGNED WORD	WORD
SIGNED LONGWORD	LONG
F_FLOATING	SINGLE
D_FLOATING	DOUBLE
G_FLOATING	GFLOAT
PACKED DECIMAL	DECIMAL

If a CDD/Repository data definition containing an unsupported data type is extracted, HP BASIC signals the informational message "Datatype in CDD/Repository not supported, substituted group for: <field-name>" and translates the data type by creating a group to contain the data type field. The group name is the name of the unsupported data type followed by the text "_VALUE". This allows you to access the field name within the group.

An example of how HP BASIC translates unsupported CDD/Repository data types is shown in the following CDDL data definition and corresponding HP BASIC RECORD statement:

CDDL Definition

```
define record CDD$top.basic.stock
        description is
           /* this is an example data definition that contains
           data types not supported by HP BASIC */.

        stock structure.
                product_no      datatype is text
                                size is 8 characters.
                date_ordered    datatype is date.
                status_code     datatype is unsigned byte.
                quantity        datatype is unsigned longword
                                aligned on longword.
                location        array 1:4
                                datatype is text
                                size is 30 characters.
                unit_price      datatype is longword.
       end stock structure.
end stock.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.STOCK"
        C1              !  This is an example data definition that contains
        C1              !  data types not supported by HP BASIC
        C1              RECORD  STOCK                       ! UNSPECIFIED
        C1                STRING  PRODUCT_NO  = 8           ! TEXT
        C1                GROUP   DATE_ORDERED              ! DATE
        C1                  STRING  STRING_VALUE  = 8
        C1                END GROUP
        C1                GROUP   STATUS_CODE               ! UNSIGNED BYTE
        C1                  BYTE    BYTE_VALUE
        C1                END GROUP
        C1                STRING  FILL = 3
        C1                GROUP   QUANTITY                  ! UNSIGNED LONGWORD
        C1                  LONG    LONG_VALUE
        C1                END GROUP
        C1                STRING  LOCATION(1 TO 4) = 30     ! TEXT
        C1                GROUP   UNIT_PRICE                ! UNSIGNED LONGWORD
        C1                  LONG    LONG_VALUE
        C1                END GROUP
        C1              END RECORD
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: STOCK::DATE_ORDERED.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: STOCK::STATUS_CODE.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: STOCK::QUANTITY.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: STOCK::UNIT_PRICE.
```

Table 21–2 describes CDD/Repository data types not supported by HP BASIC and their translation.

Table 21–2 Unsupported CDD/Repository Data Types

Data Type	HP BASIC Translation
UNSIGNED BYTE	GROUP CDD/Repository-field-name BYTE BYTE_VALUE END GROUP
UNSIGNED WORD	GROUP CDD/Repository-field-name WORD WORD_VALUE END GROUP

(continued on next page)

Table 21–2 (Cont.) Unsupported CDD/Repository Data Types

Data Type	HP BASIC Translation
UNSIGNED LONGWORD	GROUP CDD/Repository-field-name LONG LONG_VALUE END GROUP
SIGNED QUADWORD	GROUP CDD/Repository-field-name STRING STRING_VALUE = 8 END GROUP
UNSIGNED QUADWORD	GROUPCDD/Repository-field-name STRING STRING_VALUE = 8 END GROUP
SIGNED OCTAWORD	GROUP CDD/Repository-field-name STRING STRING_VALUE = 16 END GROUP
UNSIGNED OCTAWORD	GROUP CDD/Repository-field-name STRING STRING_VALUE = 16 END GROUP
H_FLOATING	GROUP CDD/Repository-field-name STRING STRING_VALUE = 16 END GROUP
F_FLOATING COMPLEX	GROUP CDD/Repository-field-name SINGLE SINGLE_R_VALUE SINGLE SINGLE_I_VALUE END GROUP
D_FLOATING COMPLEX	GROUP CDD/Repository-field-name DOUBLE DOUBLE_R_VALUE DOUBLE DOUBLE_I_VALUE END GROUP
G_FLOATING COMPLEX	GROUP CDD/Repository-field-name GFLOAT GFLOAT_R_VALUE GFLOAT GFLOAT_I_VALUE END GROUP
H_FLOATING COMPLEX	GROUP CDD/Repository-field-name HFLOAT HFLOAT_R_VALUE HFLOAT HFLOAT_I_VALUE END GROUP
ZONED NUMERIC	GROUP CDD/Repository-field-name STRING STRING_VALUE = length END GROUP

(continued on next page)

Table 21–2 (Cont.) Unsupported CDD/Repository Data Types

Data Type	HP BASIC Translation
UNSIGNED NUMERIC	GROUP CDD/Repository-field-name STRING STRING_VALUE = length END GROUP
LEFT SEPARATE NUMERIC	GROUP CDD/Repository-field-name STRING STRING_VALUE = length + 1 END GROUP
LEFT OVERPUNCHED NUMERIC	GROUP CDD/Repository-field-name STRING STRING_VALUE = length END GROUP
RIGHT SEPARATE NUMERIC	GROUP CDD/Repository-field-name STRING STRING_VALUE = length + 1 END GROUP
RIGHT OVERPUNCHED NUMERIC	GROUP CDD/Repository-field-name STRING STRING_VALUE = length END GROUP
VARYING STRING	GROUP CDD/Repository-field-name WORD WORD_VALUE STRING STRING_VALUE = length END GROUP
BIT[1]	GROUP CDD/Repository-field-name STRING STRING_VALUE = length /8 END GROUP
DATE	GROUP CDD/Repository-field-name STRING STRING_VALUE = 8 END GROUP
POINTER	GROUP CDD/Repository-field-name LONG LONG_VALUE END GROUP

[1]CDD/Repository specifies bit field length in bits; HP BASIC specifies string length in bytes. If the length in bits does not divide evenly into bytes, HP BASIC signals the error "Field <fieldname> from CDD/Repository has bit offset or length."

(continued on next page)

Table 21–2 (Cont.) Unsupported CDD/Repository Data Types

Data Type	HP BASIC Translation
UNSPECIFIED	GROUP CDD/Repository-field-name STRING STRING_VALUE = length END GROUP
VIRTUAL FIELD	Ignored

The following sections describe how HP BASIC translates CDD/Repository data types.

21.9.1 Character String Data Types

There are two CDD/Repository character string data types, TEXT and VARYING STRING. The TEXT data type translates directly into the HP BASIC STRING data type. VARYING STRING is not a supported HP BASIC data type; therefore, HP BASIC creates a group to contain the field.

The following example is a CDD/Repository definition that contains both the TEXT and VARYING STRING data types and the translated HP BASIC RECORD statement:

Example 21–1 CDDL

```
define record CDD$top.basic.strings
    description is

        /* test */.

    basicstrings structure.
        abc         datatype is text size is 10.
        xyz         datatype is varying string size is 16.
    end basicstrings structure.
end strings.
```

In the VARYING STRING data type, the actual character string is preceded by a 16-bit count field. Therefore, HP BASIC creates a WORD variable to hold the specified string length.

Example 21–2 Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.STRINGS"
        C1              !   test
        C1              RECORD  BASICSTRINGS              ! UNSPECIFIED
        C1                STRING  ABC  = 10               ! TEXT
        C1                GROUP   XYZ                     ! VARYING STRING
        C1                  WORD    WORD_VALUE
        C1                  STRING  STRING_VALUE  = 16
        C1                END GROUP
        C1              END RECORD
.................1
%BASIC-I-CDD/SUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: BASICSTRINGS::XYZ.
```

Note

The count field preceding the VARYING STRING is actually an UNSIGNED WORD. Therefore, the count field of a VARYING STRING whose length is greater than 32,767 is interpreted by HP BASIC as a negative number.

In the previous example, the group name (XYZ) is the same name as a CDD/Repository field. Therefore, HP BASIC supplies an additional name for the RECORD components. The supplied names are WORD_VALUE and STRING_VALUE. For example, the following program statement creates an instance of the record *BASICSTRINGS*, called *MY_REC*:

```
100      MAP (TEST) BASICSTRINGS MY_REC
```

The names you use to reference these components in HP BASIC are MY_REC::XYZ::WORD_VALUE and MY_REC::XYZ::STRING_VALUE.

21.9.2 Integer Data Types

CDD/Repository refers to integer data types as fixed-point data types. CDD/Repository supports BYTE, WORD, LONGWORD, QUADWORD, and OCTAWORD integer data types. Each of these data types can have the following additional attributes:

SIGNED
UNSIGNED
SIZE
DIGITS
FRACTION

BASE
SCALE

In CDDL, if integer data types are not specified as being signed or unsigned, the default is unsigned. HP BASIC supports only signed BYTE, signed WORD, signed LONGWORD, and signed QUADWORD integers. If a CDD/Repository data definition containing an unsigned BYTE, WORD, LONGWORD, or QUADWORD integer is extracted, HP BASIC signals the informational message "Datatype in CDD/Repository not supported, substituted group for: <field-name>," and creates a group to contain the field. Because the group name is the same as the CDD/Repository field name, HP BASIC assigns a new name to the field. This is shown in the following CDDL data definition and corresponding HP BASIC RECORD statement:

CDDL Definition

```
define record CDD$top.basic.integers
    description is

       /*Test of selected integer data types*/.
    basicint structure.
        my_byte         datatype is signed byte.
        my_ubyte        datatype is byte.
        my_word         datatype is signed word.
        my_uword        datatype is unsigned word.
        my_long         datatype is signed longword.
        my_ulong        datatype is unsigned longword.
    end basicint structure.
end integers.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.INTEGERS"
        C1              !  Test of selected integer data types
        C1              RECORD  BASICINT                    ! UNSPECIFIED
        C1                BYTE    MY_BYTE                   ! SIGNED BYTE
        C1                GROUP   MY_UBYTE                  ! UNSIGNED BYTE
        C1                  BYTE    BYTE_VALUE
        C1                END GROUP
        C1                WORD    MY_WORD                   ! SIGNED WORD
        C1                GROUP   MY_UWORD                  ! UNSIGNED WORD
        C1                  WORD    WORD_VALUE
        C1                END GROUP
        C1                LONG    MY_LONG                   ! SIGNED LONGWORD
        C1                GROUP   MY_ULONG                  ! UNSIGNED LONGWORD
        C1                  LONG    LONG_VALUE
        C1                END GROUP
        C1              END RECORD
.................1
```

```
%BASIC-I-CDDSUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: BASICINT::MY_UBYTE.
%BASIC-I-CDDSUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: BASICINT::MY_UWORD.
%BASIC-I-CDDSUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: BASICINT::MY_ULONG.
```

When the previous data definition is extracted from CDD/Repository, HP BASIC signals an informational message for each of the unsigned data types, and names the CDD/Repository unsigned byte field BYTE_VALUE, the CDD/Repository unsigned word field WORD_VALUE, and the CDD/Repository unsigned longword field LONG_VALUE.

HP BASIC does not support OCTAWORD integers. If a CDD/Repository definition contains an OCTAWORD integer, HP BASIC signals the informational message "Datatype in CDD/Repository not supported, substituted group for: <field-name>" and creates a group to contain the field and a string component within the group. The string component is 16 bytes for OCTAWORD integers. For example:

CDDL Definition

```
define record CDD$top.basic.bigintegers
    description is

        /*Test of quadword and octaword integer data types*/.

    basicint structure.
        my_quad         datatype is signed quadword.
        my_octa         datatype is signed octaword.
    end basicint structure.
end bigintegers.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.BIGINTEGERS"
        C1              !  Test of quadword and octaword integer data types
        C1              RECORD  BASICINT                  ! UNSPECIFIED
        C1                QUAD   MY_QUAD                  ! SIGNED QUADWORD
        C1                GROUP   MY_OCTA                 ! SIGNED OCTAWORD
        C1                  STRING  STRING_VALUE  = 16
        C1                END GROUP
        C1              END RECORD
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
                substituted group for: BASICINT::MY_OCTA.
```

CDD/Repository supports the SCALE keyword to specify an implied exponent in integer data types, and the BASE keyword (supported in CDDL only) to specify that the scale for a fixed-point field is to be interpreted in a numeric base other than 10. HP BASIC does not support these integer attributes. Therefore, HP BASIC signals the informational message "CDD/Repository specifies SCALE for <name>. Not supported" for fixed-point fields containing

a SCALE specification, and the error message "CDD/Repository attributes for <name> are other than base 10" for fixed-point fields specifying a base other than 10. For example:

CDDL Definition

```
define record CDD$top.basic.funnyintegers
    description is

        /* Test of quadword and octaword integer data types */.

    basicint structure.
        my_byte         datatype is signed byte scale 2.
        my_long         datatype is signed longword base 8.
    end basicint structure.
end funnyintegers.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.FUNNYINTEGERS"
        C1              !   Test of quadword and octaword integer data types
        C1              RECORD  BASICINT                    ! UNSPECIFIED
        C1                GROUP   MY_BYTE                   ! SIGNED BYTE
        C1                  BYTE    BYTE_VALUE
        C1                END GROUP
        C1                LONG    MY_LONG                   ! SIGNED LONGWORD
        C1              END RECORD
%BASIC-I-CDDATTSCA,     CDD specifies SCALE for BASICINT::MY_BYTE. Not supported
%BASIC-E-CDDATTBAS,     CDD attributes for BASICINT::MY_LONG are other than base 10
```

At compilation time, HP BASIC also signals these warning errors for each reference to fields that are not base 10 or that have a SCALE.

21.9.3 Floating-Point Data Types

CDD/Repository supports F_floating, D_floating, and G_floating data types.[1] These correspond to the BASIC SINGLE, DOUBLE, and GFLOAT data types, respectively. As with fixed-point data types, CDD/Repository also allows the specification of scale and base for floating-point data types. If a CDD/Repository data definition contains a floating-point field that specifies a SCALE or BASE, HP BASIC signals the informational message "CDD/Repository specifies SCALE for <name>. Not supported" or the error message "CDD/Repository attributes for <name> are other than base 10." For example:

[1] HP BASIC does not support the H_floating or HFLOAT data type.

CDDL Definition

```
define record floats
    description is

       /*Test of floating-point data types*/.

    basicfloat structure.
        my_single       datatype is f_floating scale 3.
        my_double       datatype is d_floating base 16.
        my_gfloat       datatype is g_floating.
    end basicfloat structure.
end floats.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.FLOATS"
        C1              !  Test of floating-point data types
        C1              RECORD  BASICFLOAT                ! UNSPECIFIED
        C1                GROUP   MY_SINGLE               ! F_FLOATING
        C1                  SINGLE  SINGLE_VALUE
        C1                END GROUP
        C1                DOUBLE  MY_DOUBLE               ! D_FLOATING
        C1                GFLOAT  MY_GFLOAT               ! G_FLOATING
        C1              END RECORD
................1
%BASIC-I-CDDATTSCA, 1:   CDD specifies SCALE for BASICFLOAT::MY_SINGLE.
                         Not supported
%BASIC-E-CDDATTBAS, 1:   CDD attributes for BASICFLOAT::MY_DOUBLE
                         are other than base 10
```

In addition, CDD/Repository supports complex floating-point numbers, but HP BASIC does not support them. Complex floating-point numbers consist of a real and an imaginary part. Each part requires the same amount of storage as a simple floating-point number. Therefore, each complex floating-point number requires twice as much storage as a simple floating-point number.

If a CDD/Repository data definition containing complex numbers is extracted, HP BASIC signals the informational message "Datatype in CDD/Repository not supported, substituted group for <field-name>," and creates a group to contain the field. As before, HP BASIC uses the data type and *_VALUE* to create the group name, but because each complex number contains both a real and an imaginary part, HP BASIC adds an "_R" to the name of the real part and an "_I" to the name of the imaginary part. This is shown in the following CDD/Repository data definition and corresponding HP BASIC RECORD statement:

CDDL Definition

```
define record CDD$top.basic.complex
    description is

        /* test complex data types */.

        complex structure.
        my_s_complex_1  datatype         f_floating_complex.
        my_d_complex_1  datatype         d_floating_complex.
        my_g_complex_1  datatype         g_floating_complex.
    end complex structure.
end complex.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.COMPLEX"
        C1              !   test complex data types
        C1              RECORD  COMPLEX               ! UNSPECIFIED
        C1                GROUP   MY_S_COMPLEX_1      ! F_FLOATING_COMPLEX
        C1                  SINGLE  SINGLE_R_VALUE
        C1                  SINGLE  SINGLE_I_VALUE
        C1                END GROUP
        C1                GROUP   MY_D_COMPLEX_1      ! D_FLOATING_COMPLEX
        C1                  DOUBLE  DOUBLE_R_VALUE
        C1                  DOUBLE  DOUBLE_I_VALUE
        C1                END GROUP
        C1                GROUP   MY_G_COMPLEX_1      ! G_FLOATING_COMPLEX
        C1                  GFLOAT  GFLOAT_R_VALUE
        C1                  GFLOAT  GFLOAT_I_VALUE
        C1                END GROUP
        C1              END RECORD
.................1
%BASIC-I-CDDSUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: COMPLEX::MY_S_COMPLEX_1.
%BASIC-I-CDDSUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: COMPLEX::MY_D_COMPLEX_1.
%BASIC-I-CDDSUBGRO, 1:        data type in CDD/Repository not supported,
                substituted group for: COMPLEX::MY_G_COMPLEX_1.
```

21.9.4 Decimal String Data Types

CDD/Repository supports the following forms of decimal string data types:

- LEFT OVERPUNCHED NUMERIC
- LEFT SEPARATE NUMERIC
- RIGHT OVERPUNCHED NUMERIC
- RIGHT SEPARATE NUMERIC
- PACKED DECIMAL

- UNSIGNED NUMERIC
- ZONED NUMERIC

HP BASIC supports only the PACKED DECIMAL decimal string data type, which corresponds to the HP BASIC DECIMAL data type. For all other decimal string data types, HP BASIC creates a group with the same name as the CDD/Repository subordinate field, and creates a string record component to contain the field. For example:

CDDL Definition

```
define record CDD$top.basic.decimalstring
    description is

        /* test decimal string data types */.

    decimalstring structure.
        my_packed_decimal          datatype is packed decimal
                                           size is 5 digits 2 fractions.
        my_zoned_numeric           datatype is zoned numeric
                                           size is 6 digits 2 fractions.
        my_unsigned_numeric        datatype is unsigned numeric
                                           size is 8 digits 4 fractions.
        my_lef_sep_numeric         datatype is left separate numeric
                                           size is 10 digits 3 fractions.
        my_left_ovpnch_numeric     datatype is left overpunched numeric
                                           size is 5 digits 2 fractions.
        my_right_sep_numeric       datatype is right separate numeric
                                           size is 3 digits 1 fractions.
        my_right_ovpnch_numeric    datatype is right overpunched numeric
                                           size is 4 digits 2 fractions.
    end decimalstring structure.
end decimalstring.
```

Translated RECORD Statement

```
     1          %INCLUDE %FROM %CDD "CDD$TOP.BASIC.DECIMALSTRING"
        C1              !  test decimal string data types
        C1              RECORD  DECIMALSTRING              ! UNSPECIFIED
        C1                DECIMAL(5 ,2 ) MY_PACKED_DECIMAL ! PACKED DECIMAL
        C1                GROUP   MY_ZONED_NUMERIC         ! ZONED NUMERIC
        C1                  STRING  STRING_VALUE  = 6
        C1                END GROUP
        C1                GROUP   MY_UNSIGNED_NUMERIC      ! UNSIGNED NUMERIC
        C1                  STRING  STRING_VALUE  = 8
        C1                END GROUP
        C1                GROUP   MY_LEF_SEP_NUMERIC       ! NUMERIC LEFT
                                                           ! SEPARATE
        C1                  STRING  STRING_VALUE  = 11
        C1                END GROUP
        C1                GROUP   MY_LEFT_OVPNCH_NUMERIC   ! NUMERIC LEFT
                                                           ! OVERPUNCHED
        C1                  STRING  STRING_VALUE  = 5
        C1                END GROUP
        C1                GROUP   MY_RIGHT_SEP_NUMERIC     ! NUMERIC RIGHT
                                                           ! SEPARATE
        C1                  STRING  STRING_VALUE  = 4
        C1                END GROUP
        C1                GROUP   MY_RIGHT_OVPNCH_NUMERIC  ! NUMERIC RIGHT
                                                           ! OVERPUNCHED
        C1                  STRING  STRING_VALUE  = 4
        C1                END GROUP
        C1              END RECORD
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: DECIMALSTRING::MY_ZONED_NUMERIC.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository  not supported,
               substituted group for: DECIMALSTRING::MY_UNSIGNED_NUMERIC.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: DECIMALSTRING::MY_LEF_SEP_NUMERIC.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: DECIMALSTRING::MY_LEFT_OVPNCH_NUMERIC.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: DECIMALSTRING::MY_RIGHT_SEP_NUMERIC.
%BASIC-I-CDDSUBGRO,           data type in CDD/Repository not supported,
               substituted group for: DECIMALSTRING::MY_RIGHT_OVPNCH_NUMERIC.
```

21.9.5 Other Data Types

CDD/Repository supports the following additional data types:

BIT
DATE
POINTER
UNSPECIFIED

VIRTUAL
ALPHABETIC

HP BASIC does not support these data types. HP BASIC translates these data types by signaling the informational message “Datatype in CDD/Repository not supported, substituted group for: <field name>”, and creates a group to contain the field. See Table 21–2 for a description of how HP BASIC translates these data types.

If you extract a CDD/Repository definition that contains a BIT field, the field must be a multiple of 8 bits (1 byte). This means that the following field must be aligned on a byte boundary. If the following field is not aligned on a byte boundary, HP BASIC signals the error “Field <name> from CDD/Repository has bit offset or length.”

22

Using DECwindows Motif Bindings with BASIC

This chapter explains the BASIC language exceptions for using standard DECwindows Motif Bindings. For more information about programming DECwindows Motif, see the *DECwindows Motif Guide to Application Programming*.

22.1 Overview of DECwindows Motif Concepts

This section introduces DECwindows Motif concepts. DECwindows Motif is an X Window System type of operating environment. DECwindows Motif is used on a workstation, where several windows can be displayed with different applications on each window.

To program in the DECwindows Motif environment, DECwindows Motif bindings are used to help write programs that create and manage the different resources needed to control the windowing environment.

22.2 Using DECwindows Motif Bindings with BASIC

The DECwindows Motif bindings consist of constant definitions, global variable declarations, record structures, and function prototypes. The bindings include everything that is needed to do windows programming using the DECwindows Motif Application Programming Interface (API).

The BASIC implementation of the DECwindows Motif bindings allow you to write to either the C version or the non-C version of the bindings. In either case, you will want to refer to the *VMS DECwindows User Interface Language Reference Manual* before you start programming. For information about using non-C bindings, see the *DECwindows Motif for OpenVMS Guide to Non-C Bindings*.

BASIC$HELLOMOTIF.BAS, and BASIC$HELLOBURGER.BAS supplied on the kit as examples of using the BASIC language for windows programming.

A BASIC user can code to the standard C DECwindows Motif bindings with the following exceptions:

- Any identifiers longer than 31 characters must be truncated to 31 characters. Known instances include:

  S_ _XmTraverseObscuredCallbackStru (33)
  S_XmOperationChangedCallbackStruc (33)
  S_XmDragDropFinishCallbackStruct (32)

- Any identifiers beginning with an underscore must have the underscore dropped. Known instances include:

  _DXmPrintFormatStruct
  _DXmPrintOptionMenuStruct
  _XA_MOTIF_BINDINGS
  _XA_MOTIF_WM_FRAME
  _XA_MOTIF_WM_HINTS
  _XA_MOTIF_WM_INFO
  _XA_MOTIF_WM_MENU
  _XA_MOTIF_WM_MESSAGES
  _XA_MOTIF_WM_OFFSET
  _XA_MWM_HINTS
  _XA_MWM_INFO
  _XA_MWM_MESSAGES
  _XmSDEFAULT_BACKGROUND
  _XmSDEFAULT_FONT
  _XmSecondaryResuourceDataRec
  _XmTraverseObscuredCallbackStru
  _XtCheckSubclassFlag
  _XtIsSubclassOf

- The following list of identifiers are used either as both a data type and a field name or as a BASIC keyword. They cannot be used as is, but must have the suffix _D appended when used as a data type and the suffix _F appended when used as a field name.

  dimension
  display
  font
  name
  pixel
  screen
  size
  status
  substitution

time
value
window

The DECW$MOTIF.BAS Motif bindings file includes the file DECW$MOTIF_DEFS.BAS, which contains data type aliases. This makes separate compilation of Motif application subroutines simpler. To separately compile a Motif application routine, add both of the following:

- %INCLUDE DECW$MOTIF_DEFS.BAS before the subroutine statement
- %INCLUDE DECW$MOTIF.BAS after it

22.3 DECwindows Motif Programming Examples Using BASIC

DECW$EXAMPLES contains two examples of DECwindows Motif applications in BASIC: BASIC$HELLOMOTIF.BAS and BASIC$MOTIFBURGER.BAS. SYS$LIBRARY:DECW$MOTIF.BAS, which contains the DECwindows Motif declarations, is required to build the programs. The steps to build and run the HELLOMOTIF example are:

1. Copy the needed files into your current directory:

```
$ COPY DECW$EXAMPLES:BASIC$HELLOMOTIF.* *.*
```

2. Build the Resource (UID) file:

```
$ UIL/MOTIF BASIC$HELLOMOTIF.UIL
```

3. Compile and link the BASIC program:
   - Use the following example for DECWindows Motif V1.1:

```
$ BASIC BASIC$HELLOMOTIF
$ LINK BASIC$HELLOMOTIF,SYS$INPUT/OPTIONS
SYS$LIBRARY:DECW$DXMLIBSHR.EXE/SHARE
SYS$LIBRARY:DECW$XMLIBSHR.EXE/SHARE
SYS$LIBRARY:DECW$XTSHR.EXE/SHARE
^Z
$
```

   - Use the following example for DECWindows Motif V1.2:

```
$ BASIC BASIC$HELLOMOTIF
$ LINK BASIC$HELLOMOTIF,SYS$INPUT/OPTIONS
SYS$LIBRARY:DECW$DXMLIBSHR12.EXE/SHARE
SYS$LIBRARY:DECW$MRMLIBSHR12.EXE/SHARE
SYS$LIBRARY:DECW$XMLIBSHR12.EXE/SHARE
SYS$LIBRARY:DECW$XTLIBSHRR5.EXE/SHARE
^Z
$
```

You may want to create an options file with the previously mentioned shareable libraries in it.

4. If you are not running on a workstation, make sure that your display is set correctly, for example:

   ```
   $ SET DISPLAY/CREATE/NODE=xxxx
   ```

   xxxx is the node name of a workstation with appropriate graphic capability.

5. Run the application:

   ```
   $ RUN BASIC$HELLOMOTIF
   ```

   Note

   The .UID file must be kept in the same directory as the .EXE file when run. This program looks in the current directory for the UID file.

6. Then follow the instructions in the dialog box.

22.4 Special Considerations for Handling Strings with DECwindows Motif

All strings passed between DECwindows Motif and your program must be null terminated. For example:

```
"A string" + "0"C
```

When passing a string argument to a DECwindows Motif routine, the address of the string is required. For static strings, the address of the string can easily be obtained with the LOC function. For example:

```
COMMON (c1) STRING hierarchy_file_name = 21
hierarchy_file_name = "BASIC$HELLOMOTIF.UID" + "0"C

DECLARE LONG hierarchy_file_name_array(1)
hierarchy_file_name_array(0) = LOC (hierarchy_file_name)
```

Because dynamic strings are described by a descriptor, a different means is needed to get the address of the string text. The following helper function will get the address of dynamic strings as well as static strings:

```
FUNCTION LONG ADDRESS_OF_STRING (STRING str_arg BY REF)
   OPTION TYPE=EXPLICIT, INACTIVE=SETUP
END FUNCTION (LOC (str_arg))
```

Example of passing a dynamic string to a DECwindows Motif routine:

```
DECLARE STRING temp_string
temp_string = "A string value" + "0"C
list_test = DXmCvtFCtoCS (ADDRESS_OF_STRING (temp_string),  &
     byte_count, istatus)
```

A

Compile-Time Error Messages

This appendix describes compile-time and compiler command errors, their causes, and the user action required to correct them.

A.1 Compile-Time Errors

HP BASIC diagnoses compile-time errors and does the following:

- Indicates the program line that generated the error or errors.
- Displays this program line.
- Shows you the location of the error or errors and assigns a number to each location for future reference.
- Displays the mnemonic, statement number within the line, the location number as previously displayed, and the message text. This is repeated for each error in the line.

HP BASIC repeats this procedure for each error diagnosed during compilation. The error message format for compile-time errors is:

%BASIC–<l>–<mnemonic>, <n>: <message>

<l>
Is a letter indicating the severity of the error. The severity indicator can be one of the following:

- I —- indicating information
- W —- indicating a warning
- E —- indicating an error
- F —- indicating a severe error

<mnemonic>
Is a 3- to 9-character string that identifies the error. Error messages in this appendix are alphabetized by this mnemonic.

<n>:
Is the *n*th error within the line's picture.

<message>
Is the text of the error message.

For example:

```
Diagnostic on source line 1, listing line 1, BASIC line 10

        10 DECLARE REAL BYTE A, A
.......................1.......2

%BASIC--E--CONDATSPC, 1:   conflicting data type specifications
%BASIC--E--ILLMULDEF, 2:   illegal multiple definition of name A
```

This display tells you that two errors were detected on line 10; HP BASIC displays the line containing the error, then prints a picture showing you where the errors were detected. In the example, the picture shows a 1 under the keyword BYTE and a 2 under the second occurrence of variable A. The following line shows you:

- The error mnemonic CONDATSPC
- Which error in the line's picture is referred to by the mnemonic
- The message associated with that error

In this case, the error message tells you that there are two contradictory data-type keywords in the statement. The next line shows you the same type of information for the second error; in this case, the compiler detected multiple declarations of variable A.

If a compilation causes an error of severity I or W, the compilation continues and produces an object module. If a compilation causes an error of severity E, the compilation continues but produces no object module. If a compilation causes an error of severity F, the compilation aborts immediately.

The following is an alphabetized list of compilation error messages:

ACTARGMUS, actual argument must be specified

Explanation: ERROR —- A DEF function reference contains a null argument, for example, FNA(1,,2).

User Action: Specify all arguments when referencing a DEF function.

ALLOCSML, allocated area may be too small for section

Explanation: WARNING—A MAP or COMMON with the same name exists in more than one program module, and the first one encountered by the compiler is smaller than the subsequent ones.

User Action: HP BASIC first allocates MAP and COMMON areas in the main program, then MAP and COMMON areas in subprograms, in the order in which they were loaded. Thus, you can avoid this error by loading modules with the largest MAP or COMMON first. However, it is better practice to make MAP and COMMON areas equal in size.

AMBRECCOM, ambiguous RECORD component

Explanation: ERROR—The program contains an ambiguous RECORD component reference, for example, A::D when both A::B::D and A::C::D exist.

User Action: Remove the ambiguity by fully specifying the record component.

AMPCONILL, & continuation is illegal after %INCLUDE directive

Explanation: ERROR—A program contains a %INCLUDE directive followed by an ampersand continuation to another statement. For example, the following is illegal:

```
2300  %INCLUDE %FROM %CDD "CDD$TOP.PERSONNEL.EMPLOYEE"  &
      GOTO 3000
```

Ampersand continuation of the %INCLUDE directive is legal, however.

User Action: Recode to eliminate the line continuation or use backslash continuation.

AMPCONREP, & continuation is illegal after %REPORT directive

Explanation: ERROR—A program contains a %REPORT directive followed by an ampersand continuation to another statement. For example, the following is illegal:

```
2300  %REPORT %DEPENDENCY "CDD$TOP.PERSONNEL.EMPLOYEE.COURSE_FORM"  &
      GOTO 3000
```

Ampersand continuation of the %REPORT directive itself is legal, however.

User Action: Recode to eliminate the line continuation or use backslash continuation.

ANSDEFMUS, ANSI DEF must be defined before reference

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains a reference to a DEF function before the function definition.

User Action: Renumber the line containing the function definition so that the definition precedes all references to the function.

ANSILNREQ, a line number is required on first line for ANSI

Explanation: ERROR—When you specify the /ANSI qualifier, a program must have a line number on the first line for the ANSI qualifier.

User Action: Supply a line number on the first line.

ANSKEYSPC, keywords must be delimited by spaces in /ANSI

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains a line where two elements (two keywords, a keyword and a line number, or a keyword and a string constant) are not separated by at least one space. For example, PRINT“Hello”.

User Action: Delimit all keywords, line numbers, and string constants with at least one space.

ANSLINDIG, ANSI line number may not exceed 4 digits

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains a line number with more than 4 digits, that is, a number greater than 9999.

User Action: Renumber the program lines so that no line number exceeds 9999.

ANSLINNUM, ANSI line numbers must begin in column 1

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains a line number preceded by one or more spaces or tabs.

User Action: Remove any spaces and tabs that precede the line number.

ANSREQREA, ANSI requires REAL default type

Explanation: ERROR—The /ANSI_STANDARD qualifier conflicts with the /TYPE_DEFAULT qualifier.

User Action: Do not specify a default data type other than REAL. REAL is the default.

ANSREQSCA, ANSI requires SCALE 0

Explanation: ERROR—The /ANSI_STANDARD qualifier conflicts with the /SCALE qualifier.

User Action: Do not specify a scale factor.

ANSREQSET, ANSI requires SETUP

Explanation: ERROR—The /ANSI_STANDARD qualifier conflicts with the /NOSETUP qualifier.

User Action: Do not specify /NOSETUP.

ANYDIMNOT, dimension checking not allowed on ANY

Explanation: ERROR—Both a data type of ANY and a DIM clause were specified in an EXTERNAL statement.

User Action: Remove the DIM clause from the EXTERNAL statement. ANY implies either scalar or array.

ANYNOTALL, ANY not allowed on EXTERNAL PICTURE

Explanation: ERROR—An attempt was made to specify the ANY keyword on an EXTERNAL PICTURE declaration. This is not allowed because the ANY data type should be used for calling non-BASIC procedures only.

User Action: Remove the ANY keyword from the EXTERNAL PICTURE declaration.

APPMISNUM, append file missing line number on first line

Explanation: ERROR—An attempt was made to append a source file that does not contain a line number on the first line.

User Action: Put a line number on the first line of the appended file.

APPNOTALL, append not allowed on programs without line numbers

Explanation: ERROR—The APPEND command cannot be used on a program without line numbers.

User Action: Use an include file.

ARESTYMUS, area style must be “HOLLOW”, “SOLID”, “PATTERN”, or “HATCH”

Explanation: ERROR—You specified an invalid value in the SET AREA STYLE statement.

User Action: Specify one of the values listed in the message.

AREREQTHR, AREA output requires at least 3 X,Y points

Explanation: ERROR—An AREA graphic output statement specifies less than 3 points.

User Action: Specify at least 3 points in the AREA graphic output statement.

ARGERR, illegal argument for command

Explanation: ERROR—An argument was entered for a command that does not take an argument, or an invalid argument was entered for a command, for example, SCALE A or LIST A.

User Action: Reenter the command with the proper arguments.

ARRMUSHAV, array must have 1 dimension

Explanation: ERROR—An array with multiple dimensions is specified where a one-dimensional array is required.

User Action: Specify an array that has 1 dimension.

ARRMUSELE, array must have at least 4 elements

Explanation: ERROR—You specified an array with less than four elements. This statement requires an array with at least four elements in it.

User Action: Supply an array declared as having at least 4 elements.

ARRNAMREQ, array names only allowed

Explanation: ERROR—The type of variable name required must be an array name.

User Action: Change the variable name to an array name.

ARRNOTALL, array <name> not allowed in DEF declaration

Explanation: ERROR—The parameter list for a DEF function definition contained an entire array.

User Action: Remove the array specification. Passing an entire array as a parameter to a DEF function is not allowed.

ARRTOOBIG, named array <array-name> is too large

Explanation: ERROR—An array must occupy fewer than (2^16 —- 1) bytes of storage.

User Action: Reduce the size of the array. If the array is within a record, the maximum size of the array is 65,535 bytes.

ATROVRVAR, attributes of overlaid variable <name> don't match

Explanation: ERROR—A variable name appears in more than one overlaid MAP; however, the attributes specified for the variable are inconsistent.

User Action: If the same variable name appears in multiple overlaid MAPs, the attributes (for example, data type) must be identical.

ATRPRIREF, attributes of prior reference to <name> don't match

Explanation: WARNING—A variable or array is referenced before the MAP that declares it. The attributes of the referenced variable do not match those of the declaration.

User Action: Make sure that the variable or array has the same attributes in both the reference and the declaration.

ATTGTRZER, graphics attribute value must be greater than zero

Explanation: ERROR—You specified a negative value when a positive value is required.

User Action: Supply a value greater than zero.

BADFMTSTR, invalid PRINT USING format string

Explanation: ERROR—The PRINT USING format string specified is not valid.

User Action: Supply a valid PRINT USING format string.

BADLOGIC, internal logic error detected

Explanation: ERROR—An internal logic error was detected.

User Action: This error should never occur. Please submit a Software Performance Report with a machine-readable copy of the source program.

BADNO, qualifier <name> does not accept 'NO'

Explanation: ERROR—A qualifier that does not allow a NO prefix was entered. For example, NODOUBLE.

User Action: Select the proper qualifier.

BADPROGNM, error in program name

Explanation: ERROR—The program name is longer than 39 characters or contains invalid characters.

User Action: Change the program name to be less than or equal to 39 characters and make sure that it contains only letters, digits, dollar signs, and underscores.

BADVALUE, <text> is an invalid keyword value

Explanation: FATAL—The command supplied an invalid value for a keyword.

User Action: Supply a valid value.

BASICHLB, BASIC's HELP library is not installed on this system

Explanation: INFORMATION—A HELP command was entered and the HP BASIC HELP library was not available.

User Action: See your system manager.

BIFREQNUM, built in function requires numeric expression

Explanation: ERROR—A reference to an HP BASIC built-in function contains a string instead of a numeric expression.

User Action: Supply a numeric expression.

BIFREQSTR, built in function requires string expression

Explanation: ERROR—The program specifies a numeric expression for a built-in function that requires a string argument.

User Action: Supply a string expression for the built-in function.

BLTFUNNOT, built in function not supported

Explanation: ERROR—The program contains a reference to a built-in function not supported by this version of HP BASIC.

User Action: Remove the function reference.

BOTBOUSPE, bottom boundary must be less than the top boundary

Explanation: ERROR—In a statement that specifies a viewport or windowsize, you specified a bottom boundary that is greater than or equal to the corresponding top boundary.

User Action: Correct the bottom boundary so that it is less than the top boundary.

BOUCANNOT, bound cannot be specified for array

Explanation: ERROR—An EXTERNAL statement declaring a SUB or FUNCTION subprogram specifies bounds in an array parameter, for example:

```
EXTERNAL SUB XYZ (LONG DIM(1,2,3))
```

User Action: Remove the array parameter's bound specifications. When declaring an external subprogram, you can specify only the number of dimensions for an array parameter. For example:

```
EXTERNAL SUB XYZ (LONG DIM(,,))
```

BOUMUSTBE, bounds must be specified for array

Explanation: ERROR—The program contains an array declaration that does not specify the bounds (maximum subscript value). For example:

```
DECLARE LONG A(,)
```

User Action: Supply bounds for the declared array. For example:

```
DECLARE LONG A(50,50)
```

CANCON, can't continue

Explanation: FATAL—A CONTINUE command was typed after changes had been made to the source code.

User Action: After changes have been made to the source code, you can run the program, but you cannot continue it.

CAUNOTALL, CAUSE statement not allowed in error handler

Explanation: ERROR—A CAUSE statement is specified within an error handler.

User Action: Remove the CAUSE statement from the error handler.

CDDACCERR, CDD/Repository access error

Explanation: ERROR—CDD/Repository detected an error on an attempted CDD/Repository record extraction. HP BASIC displays the CDD/Repository error.

User Action: Take action based on the associated CDD/Repository error.

CDDACCITE, CDD/Repository error while accessing item <field-name> of record

Explanation: ERROR—CDD/Repository reported an error when accessing the field. The CDD/Repository record definition is corrupt, or there is an internal error in either HP BASIC or CDD/Repository.

User Action: If the problem is not in the CDD/Repository definition, please submit a software problem report (SPR) with the source code of a small program that produces this error.

CDDACCREC, CDD/Repository error while accessing record

Explanation: ERROR—CDD/Repository reported an error when accessing the record. The CDD/Repository record definition is corrupt or there is an internal error in either HP BASIC or CDD/Repository.

User Action: If the problem is not in the CDD/Repository definition, please submit a software problem report (SPR) with the source code of a small program that produces this error.

CDDADJBOU, adjusted bounds for dimension <number> of <array> to be zero based

Explanation: INFORMATION—CDD/Repository contains an array field with a lower bound that is not zero. HP BASIC adjusts the bound so that the array is zero based.

User Action: None.

CDDALCOFF, please submit an SPR —- CDD/Repository inconsistent with allocated offset for <field-name>

Explanation: FATAL—The offset of a field within an HP BASIC RECORD differs from the offset specified by CDD/Repository for that record.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces this error.

CDDALCSIZ, please submit an SPR —- CDD/Repository inconsistent with allocated size for <field-name>

Explanation: FATAL—The amount of storage allocated for a field in an HP BASIC RECORD differs from the amount specified by CDD/Repository for that record.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces this error.

CDDALCSPN, please submit an SPR —- CDD/Repository inconsistent with allocated span for <field-name>

Explanation: FATAL—The amount of storage allocated by an HP BASIC RECORD for an array differs from the amount specified by CDD/Repository for that record.

User Action: Please submit a software problem report (SPR) with the source code of a small program this error.

CDDAMBFLD, ambiguous field name <name> for <RECORD–name>

Explanation: ERROR—More than one CDDL structure share the same level and the same name.

User Action: Change the CDD/Repository definition so that the structures have different names.

CDDATTBAS, CDD/Repository attributes for <name> are other than base 10

Explanation: ERROR—A field in a CDD/Repository definition uses the BASE keyword. This warns you that the numeric field is not interpreted as a base 10 number.

User Action: Remove the BASE attribute in CDD/Repository or avoid using the field.

CDDATTDAT, CDD/Repository data type attribute not permitted for GROUP

Explanation: ERROR—A CDD/Repository definition specified a data type after the CDD/Repository STRUCTURE keyword. HP BASIC translates STRUCTURE to an HP BASIC RECORD or GROUP statement. These HP BASIC statements do not allow data type attributes.

User Action: Change the CDD/Repository definition.

CDDATTDIG, DIGITS attribute of <field-name> not supported for datatype

Explanation: INFORMATION—The field contains a CDD/Repository fixed-point data type that specifies the number of allowed digits. This warning tells you that HP BASIC interprets the field as BYTE, WORD, LONG, or QUAD and does not support the DIGITS attribute for this data type.

User Action: None.

CDDATTSCA, CDD/Repository specifies SCALE for <RECORD-component>. Not supported.

Explanation: INFORMATION—A field in a CDD/Repository definition uses the SCALE keyword. This warns you that the field has an implied exponent.

User Action: Remove the SCALE attribute in CDD/Repository, or avoid using the field.

CDDATTTXT, CDD/Repository TEXT attribute for group <group-name> ignored

Explanation: INFORMATION—A CDD/Repository record definition specifies a data type of TEXT for the entire record.

User Action: None. HP BASIC ignores the TEXT attribute and substitutes the UNSPECIFIED attribute.

CDDBASNAM, CDD/Repository specified BASIC name <name> has illegal form

Explanation: ERROR—The HP BASIC name specified in the CDD/Repository record definition is a reserved keyword or contains an illegal character.

User Action: Change the invalid field name.

CDDBITFLD, field <field-name> from CDD/Repository has bit offset or length

Explanation: ERROR—A CDD/Repository field does not start on a byte boundary.

User Action: Change the bit field in CDD/Repository to have a length that is a multiple of 8 bits.

CDDCOLMAJ, <array-name> from CDD/Repository is a column major array

Explanation: ERROR—An array specified in a CDD/Repository definition is column-major rather than row-major. Thus, it is incompatible with HP BASIC arrays.

User Action: Change the CDD/Repository definition to be a row-major array.

CDDDIGERR, decimal digits of <VALUE> in CDD/Repository out of range for <field-name>

Explanation: ERROR—A packed numeric CDD/Repository definition specifies more than 31 digits.

User Action: Reduce the number of digits specified in the CDD/Repository definition.

CDDDIMNOT, RECORD cannot be dimensioned

Explanation: ERROR—A CDD/Repository definition is itself an array. This is incompatible with HP BASIC RECORDs, which can contain arrays but cannot be arrays.

User Action: None. You cannot access CDD/Repository definitions that are arrays.

CDDDUPREC, RECORD <name> from CDD/Repository has duplicate name

Explanation: ERROR—The CDD/Repository record name conflicts with a previous RECORD name. The previous RECORD name may be a standard HP BASIC RECORD or another CDD/Repository record.

User Action: Remove one of the duplicate definitions.

CDDFLDNAM, field name missing

Explanation: ERROR—The CDD/Repository definition contains a field that is not named.

User Action: Supply a field name for the CDD/Repository definition.

CDDINIIGN, initial value specified in CDD/Repository ignored for: name

Explanation: INFORMATION—The specification of an initial value is unsupported by BASIC.

User Action: Set the initial value of this field in your application program.

CDDINTONLY, % not allowed on <name> with noninteger datatype

Explanation: ERROR—The % suffix is allowed only on numeric data types.

User Action: Remove the % suffix from the variable name or change the data-type keyword.

CDDLOWBOU, lower bound omitted for dimension <number> of <array-name>

Explanation: ERROR—An array in a CDD/Repository definition does not specify a lower bound.

User Action: Check to make sure the omission is not a mistake. HP BASIC supplies a lower bound of zero and continues after issuing this warning.

CDDMAXDIM, <array-name> exceeds maximum dimensions

Explanation: ERROR—An array in a CDD/Repository definition specifies more than 32 dimensions.

User Action: Reduce the number of dimensions in the CDD/Repository definition.

CDDNAMKEY, <name> is a BASIC keyword

Explanation: ERROR—A CDD/Repository definition contains a field name that is a reserved word in HP BASIC.

User Action: Change the name in the CDD/Repository definition or supply an HP BASIC name clause.

CDDOCCIGN, OCCURS DEPENDING ON clause for <array-name> from CDD/Repository ignored

Explanation: INFORMATION—CDD/Repository contains an array field with a variable number of elements. HP BASIC creates an array large enough for the maximum value.

User Action: If you modify the array field, be sure also to change the field that contains the number of array elements.

CDDOFFERR, CDD/Repository offset error, field <field-name> offsets out of order

Explanation: ERROR—The CDD/Repository definition has been corrupted or there is an internal error in either HP BASIC or CDD/Repository.

User Action: If the problem is not in the CDD/Repository definition, please submit a software problem report (SPR) with the source code of a small program that produces this error.

CDDPLUSERR, CDD/Repository access error

Explanation: ERROR—CDD/Repository detected an error while attempting to record dependency data. HP BASIC displays the CDD/Repository error.

User Action: Take action based on the associated CDD/Repository error.

CDDPREERR, decimal precision of <VALUE> in CDD/Repository out of range for <field-name>

Explanation: ERROR—The number of fractional digits for a packed decimal field is greater than the total number of digits specified for that field.

User Action: Change the number of fractional digits in CDD/Repository to be less than or equal to the total number of digits.

CDDRECFOR, CDD/Repository record format is not fixed

Explanation: ERROR—CDD/Repository supports both variable and fixed-length records. HP BASIC supports only fixed-length records.

User Action: Change the CDD/Repository record definition to specify fixed-length.

CDDRECNAM, record from CDD/Repository does not have a record name

Explanation: ERROR—HP BASIC uses the field name of the outermost structure to name the record, and therefore cannot include a CDD/Repository record that does not provide a record name.

User Action: Change the CDD/Repository record definition to provide a field name for the outermost structure of the record.

CDDSCAERR, decimal scale of <scale-factor> is out of range for <field> from CDD/Repository

Explanation: ERROR—The scale factor for a packed decimal CDD/Repository field is greater than the number of digits in the field or less than zero.

User Action: Change the scale factor in the CDD/Repository definition.

CDDSCAZER, scale 0 specified for CDD/Repository field <field-name>

Explanation: INFORMATION—A CDD/Repository field specifies no scale factor for a D_floating field, but the HP BASIC program specifies a nonzero scale factor.

User Action: Use a scale factor of zero in the HP BASIC program.

CDDSTRONLY, $ not allowed on <name> with nonstring datatype

Explanation: ERROR—The $ suffix is only allowed on string data types.

User Action: Remove the $ suffix from the variable name or change the data-type keyword.

CDDSUBGRO, substituted GROUP for <field-name>. Data type in CDD/Repository not supported.

Explanation: INFORMATION—The CDD/Repository definition specifies a data type that is not native to HP BASIC. HP BASIC creates a GROUP with the same name as the CDD/Repository field and creates variable names for the GROUP components.

User Action: None.

CDDTAGIGN, tag value ignored for <field-name> from CDD/Repository

Explanation: INFORMATION—The CDD/Repository record definition contains a VARIANTS OF.

User Action: None. HP BASIC translates the VARIANTS OF as if it were a regular variant; however, the tag value is ignored.

CDDUNSDAT, data type specified in CDD/Repository for <field-name> not supported

Explanation: ERROR—The data type specified for a field is not supported by HP BASIC.

User Action: Change the data type in the CDD/Repository record definition.

CDDUPPBOU, upper bound omitted for dimension <number> of <array-name>

Explanation: ERROR—An array in a CDD/Repository definition does not specify an upper bound.

User Action: Specify an upper bound in the CDD/Repository definition.

CDDVARFLD, field <name> from CDD/Repository has variable offset or length

Explanation: ERROR—A CDD/Repository field can be either variable or fixed-length. HP BASIC supports only fixed-length fields.

User Action: Change the CDD/Repository definition.

CHAEXPMUS, channel expression must be numeric

Explanation: ERROR—The program contains a nonnumeric channel expression, for example, PUT #A$

User Action: Change the channel expression to be numeric.

CHALINCLA, CHAIN does not support line number clause

Explanation: ERROR—A CHAIN statement contains a LINE keyword and a line number argument.

User Action: Remove the LINE keyword and the line number argument.

CHANOTALL, CHANGES not allowed on primary key

Explanation: ERROR—The PRIMARY KEY clause in an OPEN statement specifies CHANGES.

User Action: Remove the CHANGES keyword; you cannot change the value of a primary key.

CHASTAAMB, CHANGE statement is ambiguous

Explanation: ERROR—A string variable and a numeric array have the same name in a CHANGE statement.

User Action: Change the name of the string variable or the numeric array.

CLIPMUSBE, clipping must be set to “ON” or “OFF”

Explanation: ERROR—You specified an invalid value in the SET CLIP statement.

User Action: Specify one of the values listed in the message.

CLOSEIN, error closing <file-name> as input

Explanation: ERROR—An error was detected while closing an input file.

User Action: Take corrective action based on the associated message.

CLOSEOUT, error closing <file-name> as output

Explanation: ERROR—An error was detected while closing an output file.

User Action: Take corrective action based on the associated message.

CMDNOTALL, command not allowed on programs without line numbers

Explanation: ERROR—A command that cannot be used on a program without line numbers has been used on a program without line numbers.

User Action: Do not use this command on programs without line numbers.

CODLENEST, internal code length estimate error. Submit an SPR

Explanation: FATAL—HP BASIC has incorrectly estimated the size of the generated code for your program.

User Action: Submit a software problem report (SPR) with the program that caused the error. (You can often work around this error by making a simple change to your code.)

COLOUTRAN, color intensities must be in the range 0.0 to 1.0

Explanation: ERROR—The value specified for color intensity is either less than 0.0 or greater than 1.0.

User Action: Supply a value from 0.0 to 1.0.

COMMAPALI, variable <name> not aligned in COMMON/MAP <name>

Explanation: INFORMATION—In a COMMON or MAP, the total storage preceding a REAL, WORD, LONG, or QUAD numeric variable is an odd number of bytes.

User Action: None. In HP BASIC, numeric data can start on any byte boundary.

COMMAPNEQ, COMMON/MAP area sizes are not equal for section

Explanation: WARNING—A MAP or COMMON with the same name exists in more than one program module, but the size of the areas differs.

User Action: Make the size of the COMMON or MAP areas equal in size in all modules.

COMMAPOVF, COMMON/MAP <name> is too large

Explanation: ERROR—The program contains a MAP or COMMON longer than (2^31 —- 1) longwords.

User Action: Reduce the length of the COMMON or MAP.

CONCOMSYN, conditional compilation cannot be used with /SYNTAX

Explanation: FATAL—The /SYNTAX_CHECKING qualifier is in effect when a program line containing the %IF, %THEN, %ELSE, or %END %IF lexical directive was entered.

User Action: Turn off syntax checking before entering a program line containing the %IF, %THEN, %ELSE, or %END %IF lexical directive.

CONDATSPC, conflicting data type specifications

Explanation: ERROR—The program contains a declarative statement containing two or more consecutive and contradictory data-type keywords, for example, DECLARE REAL BYTE.

User Action: Remove one of the data-type keywords or make sure that the keywords refer to the same generic data type. For example, DECLARE REAL SINGLE is valid.

CONEXPREQ, constant expression required

Explanation: ERROR—A statement specifies a variable, built-in function reference or exponentiation where a constant is required.

User Action: Supply an expression containing only literals or declared constants or remove the exponentiation operation.

CONTARNOT, CONTINUE target not legal in detached error handlers

Explanation: ERROR—A CONTINUE statement within a detached WHEN block error handler contains a target.

User Action: Remove the target line number or label from the CONTINUE statement or use an attached error handler.

CONIS_INC, constant is inconsistent with the type of <name>

Explanation: ERROR—A DECLARE CONSTANT statement specifies a value that is inconsistent with the data type of the constant, for example, a BYTE value specified for a REAL constant.

User Action: Change the declaration so that the data type of the value matches that of the constant.

CONIS_NEE, <item> requires conditional expression

Explanation: ERROR—A CASE or IF keyword is immediately followed by a floating-point or string expression.

User Action: Supply a conditional expression (relational, logical, or integer).

CONLFTSID, constant <name> not allowed on left side of assignment

Explanation: ERROR—The program tries to assign a value to a user-defined constant.

User Action: Remove the assignment statement; once you have assigned a value to a declared constant, you cannot change it.

CONNOTALL, constant <name> not allowed in assignment context

Explanation: ERROR—The program tries to assign a value to a user-defined constant.

User Action: Remove the assignment statement; once you have assigned a value to a declared constant, you cannot change it.

COOMUSBE, coordinates must be within NDC space (0.0 to 1.0)

Explanation: ERROR—The value of a coordinate is either less than 0.0 or greater than 1.0.

User Action: Supply a value from 0.0 to 1.0.

CORSTAFRA, corrupted stack frame

Explanation: ERROR—An immediate mode statement was entered after a STOP statement was executed in the VAX BASIC Environment and something corrupted the stack.

User Action: Check program logic to make sure that all array references are within array bounds. This error can also be caused by loading non-BASIC object modules in the VAX BASIC Environment.

COUONLALO, COUNT clause only allowed with array LIST clause

Explanation: ERROR—A COUNT clause was found on a SET INITIAL CHOICE statement that contains a LIST clause that does not contain a string array.

User Action: Remove the COUNT clause or use the array form of the LIST clause.

COUVALCAN, COUNT value cannot be greater than array size

Explanation: ERROR—In the COUNT clause, you specified a count that is larger than the size of the array that you supplied.

User Action: Change either the COUNT value or the size of the array so that COUNT is less than or equal to the number of elements in the array.

DATTYPEXP, data type required for variable <name> with /EXPLICIT

Explanation: ERROR—A program compiled with the /TYPE=EXPLICIT qualifier declares a variable without specifying a data type.

User Action: Supply a data-type keyword for the variable or compile the program without the /TYPE=EXPLICIT qualifier.

DATTYPNOT, data type keyword not allowed in SUB statement

Explanation: ERROR—A SUB statement contains a data-type keyword between the subprogram name and the parameter list.

User Action: Remove the data-type keyword. In a SUB statement, data-type keywords can appear only within the parameter list.

DATTYPREQ, data type required in EXTERNAL CONSTANT declaration

Explanation: ERROR—An EXTERNAL CONSTANT statement has no data-type keyword.

User Action: Supply a data-type keyword to specify the data type of the external constant.

DECIMERR, DECIMAL overflow

Explanation: WARNING—The program contains a DECIMAL expression whose value is outside the valid range.

User Action: Reduce the value of the DECIMAL expression.

DECLEXSYN, DECLARED lexical function syntax error

Explanation: ERROR—The syntax of the %DECLARED lexical function is specified incorrectly.

User Action: Supply the correct syntax.

DECPREOUT, DECIMAL precision specification out of range

Explanation: ERROR—In the declaration for a packed decimal variable or constant, the number of digits to the right of the decimal point is greater than the total number of digits specified, or greater than 31.

User Action: Change the declaration so that the total number of digits specified is less than 31, and the number of digits to the right of the decimal point is less than or equal to the total number of digits.

DECSIZOUT, DECIMAL size specification out of range

Explanation: ERROR—The declaration for a packed decimal variable or variable specifies more than 31 digits.

User Action: Change the declaration to specify 31 or fewer digits.

DEFEXPCOM, expression with DEF* too complex, moving <name> invocation

Explanation: WARNING—A DEF* is being invoked from within a complex expression. To simplify the expression, the compiler will evaluate the DEF*(s) first. (Alpha BASIC only.)

User Action: Rewrite statement into simpler expressions.

DEFINVNOT, DEF invocation not allowed in assignment context

Explanation: ERROR—A DEF function invocation (including a parameter list) appears on the left side of an assignment statement.

User Action: Remove the assignment statement. You cannot assign values to a function invocation.

DEFMODNOT, DEF <name> mode not as declared

Explanation: ERROR—The specified data type in a function declaration disagrees with the data type specified in the function definition.

User Action: Make the data-type specifications match in both the function declaration and the function definition.

DEFNOTDEF, DEF <name> not defined

Explanation: ERROR—The program contains a reference to a nonexistent user-defined function.

User Action: Define the function in a DEF statement.

DEFNOTWHE, DEF not allowed in WHEN block or handler

Explanation: ERROR—A DEF function definition is not allowed in a WHEN block or its associated handler.

User Action: Remove the DEF function definition from within the WHEN block or handler.

DEFRESREF, DEF <name> result reference illegal in this context

Explanation: ERROR—The program attempts to assign a value to a DEF name outside the DEF block.

User Action: Remove the assignment statement. You cannot assign a value to a DEF outside of the DEF block.

DEFSIZNOT, DEF <name> decimal size not as declared

Explanation: ERROR—The DECIMAL(d,s) size specified in the DEF statement does not match the DECIMAL(d,s) used in the associated DECLARE DEF statement.

User Action: Make the DECIMAL size specification agree in both the DECLARE DEF and DEF statements.

DEFSTAPAR, DEF* formal <formal-name> inconsistent with usage outside DEF*

Explanation: ERROR—A DEF* formal parameter has the same name as a program variable, but different attributes.

User Action: You should not use the same names for DEF* parameters or program variables. If you do, you must ensure that they have the same data type and size.

DEFSTOOCMPX, DEF* <name> too complex to compile

Explanation: ERROR—A DEF* function uses too many temporary variables or parameters, or is too complex to compile.

User Action: Simplify and/or break the function into smaller pieces.

DEFSTRPAR, DEF string parameter is illegal in MAP DYNAMIC or REMAP

Explanation: ERROR—You cannot use a static string that is a parameter declared in a DEF or DEF* function as the storage area in a MAP DYNAMIC or REMAP statement.

User Action: Change the storage area specification in the MAP DYNAMIC or REMAP statement to use either a MAP name or a static string variable that is not a parameter to the DEF or DEF* function.

DELETE, ignoring <item>

Explanation: ERROR—The program contains a syntax error. The compiler tries to recover from the error by ignoring an operator or separator in the source line. For example, DIM A(3,) is a syntax error, but HP BASIC continues the compilation by ignoring the comma. The compilation continues only in order to discover other errors; no object module is produced.

User Action: Correct the syntax error in the displayed line.

DEPNOTANS, /DEPENDENCY_DATA qualifier not allowed with /ANSI

Explanation: ERROR—The /DEPENDENCY_DATA qualifier conflicts with the /ANSI_STANDARD qualifier.

User Action: Specify either the /DEPENDENCY_DATA qualifier or the /ANSI_STANDARD qualifier, but not both.

DESCOMABORT, /DESIGN=COMMENT processing has been aborted due to an internal error—please submit an SPR

Explanation: INFORMATION—The compiler was unable to process comment information due to an internal error.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces the error.

DESCOMERR, error in processing design information

Explanation: WARNING—The design information was syntactically incorrect.

User Action: You should respecify the design information and compile the program again.

DESIGNTOOOLD, /DESIGN=COMMENT processing routines are too old for the compiler

Explanation: WARNING—The compiler encountered obsolete routines.

User Action: Install a new version of the Language Sensitive Editor for OpenVMS.

DESOUTRAN, destination out of range

Explanation: FATAL—The branch destination in an ON GOSUB statement is greater than 32,767 bytes away from the statement.

User Action: Reduce the distance between the destination and the statement.

DIMOUTRAN, dimension is out of range

Explanation: ERROR—The program contains the declaration of an array that specifies a negative number as a dimension.

User Action: Change the dimension to a positive number.

DIMLSSZERO, dimension must be greater than zero

Explanation: ERROR—The number specified for a dimension must be greater than zero.

User Action: Change the number to be greater than zero.

DIMTOOBIG, dimension for array <name> must be between 1 and <number>

Explanation: ERROR—The number of the dimension specified is greater than the number of dimensions in the array.

User Action: Change the dimension number to be less than or equal to the number of dimensions in the array.

DIRMUSTBE, directive must be only item on line

Explanation: ERROR—The program contains a compiler directive that is not the only item on the line.

User Action: Place the directive on its own line.

DIRNOTIMM, directive not valid in immediate mode

Explanation: ERROR—A compiler directive was typed in the VAX BASIC Environment.

User Action: None. Compiler directives are invalid in immediate mode.

DIVBY_ZER, division by zero

Explanation: WARNING—The value of a number divided by zero is indeterminate.

User Action: Change the expression so that no expression is divided by the constant zero.

DRAWITREQ, DRAW WITH clause requires 4X4 matrix

Explanation: ERROR—A user matrix is specified in a DRAW statement WITH clause where a two-dimensional matrix with lower bounds 0 and upper bounds 4 in both dimensions is required.

User Action: Declare the matrix to be a two-dimensional matrix with lower bounds 0 and upper bounds 4 in both dimensions.

DUPCLASPE, duplicate clause specified

Explanation: ERROR—A duplicate clause was found on a SET INITIAL statement or a graphics input statement.

User Action: Remove the duplicate clause.

DUPLINNOT, duplicate line numbers not ANSI

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier from the DCL command level, or called into the VAX BASIC Environment with the OLD command while the /ANSI_ STANDARD qualifier is in effect, contains two identical line numbers.

User Action: Remove one instance of the duplicate line number. Even if you compile the program without the /ANSI_STANDARD, HP BASIC will ignore all statements connected with the first instance of the duplicate line number before compiling the program.

DUPLNFND, duplicate line number <number> found

Explanation: INFORMATION or WARNING

Explanation: INFORMATION—A line number in an include file is the same as a line number in the main source file.

Explanation: WARNING—There are two lines in the main source file with the same line number. HP BASIC keeps the second occurrence of the line number.

User Action: Correct the source by changing one of the line numbers to an unused number.

DYNATTONL, DYNAMIC attribute only valid for MAP areas

Explanation: ERROR—A COMMON keyword is followed by the DYNAMIC keyword.

User Action: Remove the DYNAMIC keyword. The DYNAMIC attribute is valid only for MAP areas.

DYNSTRINH, dynamic string variable <name> inhibits optimization

Explanation: INFORMATION—This error is reported only when the /NOSETUP qualifier is in effect. The program contains a dynamic string variable. This prevents optimization of the compiler-generated code.

User Action: Place the string variable in a COMMON keyword or MAP area.

ELENOALGN, Elements within array <array-name> are not naturally aligned.

Explanation: WARNING—Identifies record arrays that may not be naturally aligned because the size of each record element is not a multiple of the record's natural alignment. This error is reported only when the /WARNING=ALIGNMENT qualifier is in effect.

User Action: Modify the size of the record to be a multiple of the record's natural alignment.

ELSIMPCON, ELSE appears in improper context, ignored

Explanation: ERROR—The program contains an ELSE clause that either is not preceded by an IF statement or that appears after an IF has been terminated with a line number or END IF.

User Action: Remove either the ELSE clause or the terminating line number or END IF.

EMPTYOBJ, Empty object file due to error

Explanation: INFORMATION—The compiler has detected errors and therefore did not produce an object file.

User Action: The errors must be corrected before the compiler will produce an object file.

ENDIMPCON, END IF appears in improper context, ignored

Explanation: ERROR—The program contains an END IF statement that either is not preceded by an IF statement or occurs after an IF has been terminated by a line number.

User Action: Supply an IF statement or remove the terminating line number.

ENDSTAREQ, END statement required in ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier does not contain an END statement.

User Action: Include an END statement as the last statement in the program. ANSI Minimal BASIC requires an END statement.

ENTARRFIE, entire array field of virtual record cannot be passed

Explanation: ERROR—The program attempts to pass an entire array as a parameter to a subprogram when:

- The array is an item in a record
- The record is itself dimensioned as a virtual array

User Action: Assign the values of the array to another array that is of the same data type and dimension but that is not a field of a virtual array record, and pass the second array as the parameter.

ENTARRNOT, entire array not allowed in this context

Explanation: ERROR—The program specifies an entire array in a context that permits only array elements, for example, in a PRINT statement.

User Action: Remove the reference to the entire array.

ENTGRONOT, entire GROUP or RECORD not allowed in this context

Explanation: ERROR—The program specifies an entire GROUP or RECORD in a context that permits only GROUP or RECORD components, for example, PRINT ABC::XYZ where XYZ is a GROUP.

User Action: Remove the reference to the entire GROUP or RECORD.

ENTVIRARR, entire virtual array cannot be a parameter

Explanation: ERROR—The program attempts to pass an entire virtual array as a parameter.

User Action: None. You cannot pass an entire virtual array as a parameter.

EOLNOTTER, End of line does not terminate IFs due to active blocks

Explanation: ERROR—A THEN or ELSE clause contains a loop block, and a line number terminates the if-then-else before the end of the loop block.

User Action: Make sure that any loop is entirely contained in the THEN or ELSE clause.

ERLNOTALL, ERL statement not allowed in programs without line numbers

Explanation: ERROR—An ERL statement has been found in a program without line numbers.

User Action: Remove the ERL statement.

ERRACCLIB, error accessing module <mod-name> in text library <text-lib-name>

Explanation: ERROR—HP BASIC found an unexpected LIBRARIAN error while trying to %INCLUDE a text library module. This error message is followed by a specific LIBRARIAN (LBR) message.

User Action: Take appropriate action based on the associated LBR message.

ERRCLOLIB, error closing text library <text-lib-name>

Explanation: ERROR—The text library specified in a %INCLUDE directive could not be closed. This error message is followed by the specific LIBRARIAN (LBR) error.

User Action: Take appropriate action based on the associated LBR message.

ERROPEFIL, error opening file

Explanation: ERROR—The file specified in a %INCLUDE directive could not be opened. This error message is followed by the specific RMS error.

User Action: Take appropriate action based on the associated RMS error.

ERROPELIB, error opening text library <text-lib-name>

Explanation: ERROR—The text library specified in a %INCLUDE directive could not be opened. This error message is followed by the specific LIBRARIAN (LBR) error.

User Action: Take appropriate action based on the associated LBR message.

ERRREADFIL, error reading file <file_name>

Explanation: ERROR—the compiler encountered problems while reading either a BASIC source file or a CDD audit file (as specified using the /AUDIT qualifier).

User Action: Examine the secondary message that follows this message to find out what went wrong, then take the appropriate action.

ERRRECCOM, erroneous RECORD component

Explanation: ERROR—The program contains an erroneous record component, for example, specifying A::B when RECORD A has no component named B.

User Action: Remove the erroneous reference.

EXEDIMILL, executable DIMENSION illegal for static array

Explanation: ERROR—A DIMENSION statement names an array already declared with a DECLARE, COMMON, MAP, or RECORD statement, or one that was declared statically in a previous DIMENSION statement.

User Action: Remove the executable DIMENSION statement or originally declare the array as executable in a DIMENSION statement.

EXPDECREQ, explicit declaration of <name> required

Explanation: ERROR—The program is compiled with the /TYPE:EXPLICIT qualifier in effect, and the program references a variable, constant, function, or subprogram name that is not explicitly declared.

User Action: Explicitly declare the variable, constant, function, or subprogram.

EXPIFDIR, expecting IF directive

Explanation: ERROR—The program contains a %END that is not immediately followed by a %IF.

User Action: Supply a %IF immediately following the %END.

EXPNOTALL, expression not allowed in this context

Explanation: ERROR—The program contains an expression in a context that allows only simple variables, array elements, or entire arrays, for example, in FIELD and MOVE statements.

User Action: Remove the expression.

EXPNOTRES, expression does not contribute to result of string concatenation

Explanation: INFORMATION—The compiler has detected an expression that is not needed in determining a result.

User Action: Review the program to determine if the expression can be eliminated. You may want to remove the expression if it is determined to be unnecessary.

EXPTOOCOM, expression too complicated

Explanation: ERROR—The program contains an expression or statement too complicated to compile. This message can occur whenever HP BASIC is unable to allocate sufficient registers.

User Action: Recode as required; for example, rewrite the statement as two or more less complicated statements.

EXPUNAOPE, expecting unary operator or legal lexical operand

Explanation: ERROR—A compiler directive contains an invalid lexical expression, for example, %IF *3% %THEN.

User Action: Correct the lexical expression.

EXTELSFOU, extra ELSE directive found

Explanation: ERROR—The program contains a %ELSE directive that is not matched with a %IF directive.

User Action: Make sure that each %ELSE is preceded by a %IF, and that each %IF contains no more than one %ELSE clause.

EXTENDIF, extra END IF directive found

Explanation: ERROR—A program unit contains a %END %IF without a preceding %IF directive.

User Action: Supply a %IF for the %END %IF.

EXTLEFPAR, extra left parenthesis in expression

Explanation: ERROR—A compiler directive contains a lexical expression with an extra left parenthesis.

User Action: Remove the extra parenthesis.

EXTNAMTOO, EXTERNAL name too long, truncating to <new–name>

Explanation: WARNING—An EXTERNAL statement names a symbol longer than 31 characters.

User Action: Shorten the symbol name to 31 characters or less.

EXTRIGPAR, extra right parenthesis in expression.

Explanation: ERROR—A compiler directive contains a lexical expression with an extra right parenthesis.

User Action: Remove the extra parenthesis.

EXTSTRVAR, EXTERNAL STRING variables not supported

Explanation: ERROR—The program contains an EXTERNAL statement that specifies an external string variable.

User Action: Remove or change the EXTERNAL statement. HP BASIC does not support external string variables.

FEANOTANS, language feature not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains a HP BASIC feature (such as a long variable name or a string array) that does not conform to the ANSI Minimal BASIC Standard. (See Chapter 5 for more information about the ANSI Minimal Standard.)

User Action: Although HP BASIC allows you to run programs with non-ANSI language features, you must remove these features if you want your program to be transportable to other ANSI Minimal BASIC compilers.

FEANOTAVA, language feature not available in Alpha BASIC

Explanation: Feature is not currently available in Alpha BASIC.

User Action: Rewrite code to work around unavailable feature.

FIEVALONL, FIELD valid only for dynamic string variables

Explanation: ERROR—A FIELD statement contains a numeric or fixed-length string variable.

User Action: Remove the numeric or fixed-length string variable. Only dynamic string variables are valid in FIELD statements.

FILACCERR, file access error for INCLUDE directive <file-name>

Explanation: ERROR—The file named in the %INCLUDE directive was correctly opened but could not be read for some reason, for example, the disk drive was switched off line.

User Action: Take action based on the associated RMS error messages.

FILEWRITE, <prog–name> written to file: <file-name>

Explanation: INFORMATION—The specified program name has been saved in file-name.

User Action: None.

FILNOTALL, FILL not allowed in MAP DYNAMIC

Explanation: ERROR—A MAP DYNAMIC statement contains a FILL item.

User Action: Remove the FILL item.

FILNOTDEL, error deleting <file-name>

Explanation: ERROR—An error was detected in attempting to delete a file.

User Action: Supply a valid file specification, or take corrective action based on the associated message.

FILTOOBIG, FILL number <n> in overlay <m> of MAP <name> too big

Explanation: ERROR—A FILL string length or repeat count caused the compiler to try to allocate more than 2^31 longwords of storage.

User Action: Check the specified MAP statement and change the FILL string length or repeat count.

FLDNOALGN, FIELD <field-name> within RECORD <record_name> is not naturally aligned.

Explanation: WARNING—Identifies a field within a record that was found not to be naturally aligned. This error is reported only when the /WARNING=ALIGNMENT qualifier is in effect.

User Action: Modify the record so that all fields are naturally aligned.

FLOCVTILL, floating CVT valid only for SINGLE and DOUBLE

Explanation: ERROR—A CVTF$ or CVT$F function names a GFLOAT, HFLOAT, SFLOAT, TFLOAT, or XFLOAT value as an argument, or the default real size is one of these.

User Action: Use a SINGLE argument rather than SFLOAT. Use a DOUBLE argument rather than GFLOAT, TFLOAT, HFLOAT, or XFLOAT.

FLOPOIERR, floating point error or overflow

Explanation: WARNING—The program contains a numeric expression whose value is outside the valid range for the default floating-point data type.

User Action: Modify the expression so that its value is within the allowable range or select as the default REAL size a floating-point data type that has a greater range.

FNEWHINOT, exit from DEF while not in DEF

Explanation: ERROR—An FNEXIT or EXIT DEF statement has no preceding DEF statement.

User Action: Define the function before inserting an FNEXIT or EXIT DEF statement.

FNEWITDEF, end of DEF seen while not in DEF

Explanation: ERROR—An FNEND or END DEF statement has no preceding DEF statement.

User Action: Define the function before inserting an FNEND statement or delete the FNEND statement.

FORFEEMUS, FORM FEED must appear at end of line

Explanation: INFORMATION—A form-feed character is followed by other characters in the same line.

User Action: Remove the characters following the form feed. A form feed must be the last or only character on a line.

FORPARMUS, formal parameter must be supplied for <name>

Explanation: ERROR—The declaration of a DEF, SUB, or FUNCTION routine contains the parentheses for a parameter list but no parameters.

User Action: Supply a parameter list or remove the parentheses.

FORSTRPAR, formal string parameters may not be FIELDed

Explanation: ERROR—A variable name appears both in a subprogram formal parameter list and a FIELD statement in the subprogram.

User Action: Remove the variable from the FIELD statement or the parameter list.

FOUENDWIT, found end of <block> without matching <item>

Explanation: ERROR—The program contains an END SELECT, END DEF, END FUNCTION, FUNCTIONEND, SUBEND, END SUB, or END IF without a matching SELECT, DEF, SUB, FUNCTION, or IF.

User Action: Supply a SELECT, DEF, FUNCTION, SUB, or IF to match the END <block> statement, or remove the erroneous END statement.

FOUND, found <item> when expecting <item>

Explanation: ERROR—The program contains a syntax error. HP BASIC displays the item where the error was detected, then displays one or more items that make more sense in that context. The compilation continues so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

FOUNXTWIT, found NEXT without matching WHILE or UNTIL

Explanation: ERROR—The program contains a NEXT statement without a corresponding WHILE or UNTIL statement.

User Action: Supply a WHILE or UNTIL statement or remove the erroneous NEXT statement.

FOUWITMAT, found NEXT without matching FOR

Explanation: ERROR—The program contains a NEXT <control–variable> statement without a matching FOR <control–variable> statement.

User Action: Supply a FOR statement or remove the erroneous NEXT statement.

FUNINVNOT, function invocation not allowed in assignment context

Explanation: ERROR—An external function invocation (including a parameter list) appears on the left side of an assignment statement.

User Action: Remove the assignment statement. You cannot assign values to a function invocation.

FUNNESTOO, function nested too deep

Explanation: ERROR—The program contains too many levels of function definitions within function definitions.

User Action: Reduce the number of nested functions.

FUNWHINOT, exit from FUNCTION while not in FUNCTION

Explanation: ERROR—An EXIT FUNCTION or FUNCTIONEXIT statement was found in a module that is not a FUNCTION subprogram.

User Action: Remove the EXIT FUNCTION or FUNCTIONEXIT statement.

GRAARRMUS, graphics array must be integer or real

Explanation: ERROR—The specified array has a data type other than an integer or real data type.

User Action: Declare the array with an integer or real data type.

HANNOTDEF, HANDLER not allowed in DEF

Explanation: ERROR—A HANDLER definition has been found within a DEF function definition.

User Action: Remove the HANDLER definition from inside the DEF function definition.

HANNOTFOU, error handler <name> not found

Explanation: ERROR—You did not define the HANDLER you referenced in a WHEN statement.

User Action: Define the HANDLER you reference in the WHEN statement.

HANNOTWHE, HANDLER not allowed in a WHEN block or handler

Explanation: ERROR—A detached HANDLER definition was found in a WHEN block protected region or associated handler.

User Action: Remove the HANDLER definition from within all WHEN block protected regions and associated handlers.

HANWHINOT, exit from HANDLER while not in HANDLER

Explanation: ERROR—An EXIT HANDLER statement was found while not in a HANDLER block.

User Action: Remove the EXIT HANDLER statement.

HFLOATNOTS, HFLOAT is not supported

Explanation: ERROR—HFLOAT floating-point data type is not supported by Alpha BASIC.

User Action: Remove the use of HFLOAT floating-point data type, substituting either GFLOAT, TFLOAT, or XFLOAT as appropriate.

HORJUSMUS, horizontal justification must be “LEFT”, “CENTER”, “RIGHT” or “NORMAL”

Explanation: ERROR—You specified an invalid value for the horizontal component of the SET TEXT JUSTIFY statement.

User Action: Specify one of the values listed in the message.

IDEMAYAPP, IDENT directive may appear only once per module

Explanation: WARNING—The program contains more than one %IDENT compiler directive.

User Action: Remove all but one %IDENT directive.

IDENAMTOO, IDENT directive name is too long

Explanation: WARNING—The quoted string in a %IDENT directive is too long.

User Action: Reduce the length of the string. The maximum length is 31 characters.

IF_EXPMUS, IF directive expression must be terminated by THEN directive

Explanation: ERROR—A %IF directive contains a %ELSE clause with no intervening %THEN clause.

User Action: Insert a %THEN clause.

IF_IN_INC, IF directive in INCLUDE directive needs END IF directive in same file

Explanation: ERROR—A %INCLUDE file contains a %IF but no %END %IF.

User Action: Supply a %END %IF for the %INCLUDE file.

IF_NOTTER, IF statement not terminated

Explanation: ERROR—The program contains an if-then-else statement within a block (for example, a FOR-NEXT, SELECT-CASE, or WHILE block) and the end of the block was reached before the if-then-else statement was terminated.

User Action: Check program logic to be sure if-then-else statements are terminated with a line number or an END IF statement before the end of the block is reached.

ILLALLCLA, illegal ALLOW clause <clause>

Explanation: ERROR—The program contains an ALLOW clause on a GET statement, and the file was not opened with the UNLOCK EXPLICIT clause.

User Action: Either remove the ALLOW clause from the GET statement or use the UNLOCK EXPLICIT clause in the OPEN statement.

ILLARGBP2, illegal argument count for BASIC-PLUS-2

Explanation: INFORMATION—The program contains a SUB, DEF, or EXTERNAL FUNCTION reference with more than 32 parameters. This error is reported only when the /FLAG:BP2COMPATIBILITY qualifier is in effect.

User Action: If the program must run under both HP BASIC and PDP–11 BASIC-PLUS-2, the function must have 32 or fewer parameters.

ILLARGPAS, illegal argument passing mechanism

Explanation: ERROR—The program specifies an invalid argument-passing mechanism, for example, passing strings or arrays BY VALUE, or passing an entire virtual array.

User Action: Check all elements for the proper parameter-passing mechanism.

ILLCALFUN, illegal CALL of a DECIMAL, HFLOAT or STRING function

Explanation: ERROR—You attempted to use the CALL statement to invoke either a DECIMAL, HFLOAT, or STRING function.

User Action: Invoke the function not using the CALL statement.

ILLCHA, illegal character <ASCII code>

Explanation: WARNING—The program contains illegal or incorrect characters.

User Action: Examine the program for correct usage of the HP BASIC character set and possibly delete the character.

ILLCHAEXT, illegal character <ASCII code> in external name

Explanation: ERROR—The external symbol in an EXTERNAL FUNCTION or CONSTANT declaration contains an invalid character.

User Action: Remove the invalid character. External names can use only printable ASCII characters: ASCII values in the range 32 to 126, inclusive.

ILLCHAIDE, illegal character <ASCII value> in IDENT directive

Explanation: WARNING—A %IDENT directive contains an illegal character with the reported ASCII value.

User Action: Remove the illegal character.

ILLCONTYP, illegal constant type

Explanation: ERROR—The program contains an invalid declaration, for example, DECLARE RFA CONSTANT.

User Action: Remove the invalid data type. You cannot declare constants of the RFA data type.

ILLEXTPDP, <name> is illegal as a PDP–11 external name

Explanation: INFORMATION—This error is reported only when the /FLAG=BP2COMPATIBILITY qualifier is in effect. The external name is longer than six characters or contains a non-RAD50 character.

User Action: Reduce the length of the name or remove the non-RAD50 character.

ILLFRMNAM, illegally formed name

Explanation: ERROR—The program contains an invalid user identifier (such as a variable, constant, or function name).

User Action: Change the name to comply with the rules for naming user identifiers. See the *HP BASIC for OpenVMS Reference Manual* for more information.

ILLFRMNUM, illegally formed numeric constant

Explanation: ERROR—The program contains either: 1) an invalid E-format expression, or 2) a numeric constant with a digit that is invalid in the specified radix, for example, a decimal constant containing a hexadecimal digit.

User Action: Supply a valid E-format expression or a constant that is valid in the specified radix.

ILLGOTO, can't GOTO outside current procedure

Explanation: WARNING—The target line number of an immediate mode GOTO statement is outside of the currently compiled procedure.

User Action: None. If you RUN a source file containing more than one program unit, the currently compiled program is the last program unit in the source file. If you use the OLD command to read a program into memory and load one or more object modules, then type RUN, the currently compiled procedure is the program you read into memory with OLD.

ILLIDEPDP, illegal %IDENT string for PDP–11

Explanation: INFORMATION—A %IDENT compiler directive contains a string that is invalid for PDP–11 systems. This error is issued only when the BP2 compatibility flagger is enabled.

User Action: Change the %IDENT string. The string must be from 1 to 6 characters, and must contain only RAD-50 characters.

ILLIO_CHA, illegal I/O channel

Explanation: ERROR—A constant channel expression is greater than 99, or a variable channel expression is greater than 119.

User Action: If the channel expression is a constant, change to be less than or equal to 99. A variable channel expression can be less than or equal to 119; however, channels in the range 100 to 119 are reserved for programs using LIB$GET_LUN.

ILLLINNUM, illegal line number in CHAIN

Explanation: ERROR—A CHAIN with LINE statement specifies an invalid line number. Either the number is outside the valid range, or a string expression follows the LINE keyword.

User Action: Supply an integer line number from 1 to 32,767, inclusive.

ILLLOCARG, illegal LOC argument

Explanation: ERROR—An argument to the LOC function is a constant, virtual array element, or expression.

User Action: Change the argument to the LOC function.

ILLLOONES, illegal loop nesting, expecting NEXT <VARIABLE>

Explanation: ERROR—The program contains overlapping loops.

User Action: Examine the program logic to make sure that the FOR and NEXT statements for the inside loop lie entirely within the outside loop.

ILLMATOPE, illegal matrix operation

Explanation: ERROR—The program attempts matrix division. The operation is treated as a MAT multiply, and the compilation continues.

User Action: Remove the attempted matrix division. HP BASIC does not support this operation.

ILLMCHPDP, illegal passing mechanism on PDP–11s

Explanation: INFORMATION—This error is reported only when the /FLAG=BP2COMPATIBILITY qualifier is in effect. A parameter list contains a BY clause that is invalid in PDP–11 BASIC-PLUS-2, for example, specifying BY DESC for parameters that are not entire arrays or strings.

User Action: See the *HP BASIC for OpenVMS Reference Manual* for allowable BASIC-PLUS-2 parameter-passing mechanisms.

ILLMIDLEN, illegal MID assignment length

Explanation: ERROR—The value of the length in the MID statement is either greater than the length of the string or less than or equal to zero.

User Action: Correct the length to be between 1 and the length of the string.

ILLMIDSTRT, illegal MID starting value

Explanation: ERROR—The starting value in the MID statement is less than or equal to zero.

User Action: Correct the starting value to be greater than or equal to one.

ILLMODMIX, illegal mode mixing

Explanation: ERROR—The program contains string and numeric operands in the same operation.

User Action: Change the expression so that it contains either string or numeric operands, but not both.

ILLMULDEF, illegal multiple definition of name <name>

Explanation: ERROR—The program uses the same name for:

- More than one variable
- A variable and a MAP
- A variable and a COMMON
- A MAP and COMMON

User Action: Use unique names for variables, COMMONs, and MAPs.

ILLMULOPT, OPTIONAL cannot be specified more than once

Explanation: ERROR—The OPTIONAL clause was specified more than once in the EXTERNAL statement for a single SUB or FUNCTION. This is not allowed because OPTIONAL implies that all parameters following it are optional.

User Action: Fix the EXTERNAL statement so that it has at most one OPTIONAL clause per SUB or FUNCTION.

ILLNESDEF, illegally nested DEFs

Explanation: ERROR—The program contains a DEF function block within another DEF function block.

User Action: Remove the inner DEF block. A DEF cannot contain another DEF.

ILLOPEARG, illegal operation for argument

Explanation: ERROR—The program performs an operation that is inconsistent with the data type of the arguments, for example, an arithmetic operation on variables of the RFA data type.

User Action: Remove the operation or change the data type of the arguments.

ILLOPTBAS, illegal OPTION BASE value

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains an OPTION BASE statement that specifies a value other than 0 or 1.

User Action: Change the OPTION BASE statement to specify either 0 or 1.

ILLQUACOM, illegal qualifier combination

Explanation: ERROR—In the VAX BASIC Environment, you specified an illegal combination of qualifiers, such as COMPILE/NOSHOW=CDD.

User Action: Issue the command again, using a valid combination of qualifiers.

ILLSTROPE, illegal string operator

Explanation: ERROR—The program specifies an invalid string operation, for example, A$ = B$ – C$.

User Action: Replace the invalid operator.

ILLUSAFIE, illegal usage of FIELDed variable

Explanation: ERROR—A MAT statement operates on an element of a string array that appears in a FIELD statement.

User Action: Remove the array from the MAT statement.

ILLUSEUNA, illegal use of unary operator

Explanation: ERROR—A compiler directive contains an invalid lexical expression, for example, %IF 1 NOT 2.

User Action: Correct the invalid lexical expression.

ILLWAIVAL, WAIT value must be in the range 0 to 255, inclusive

Explanation: ERROR—An integer expression was specified on a WAIT clause that is less than 0 or greater than 255.

User Action: Specify an integer expression from 0 to 255.

IMMMODOPE, immediate mode operation requires storage allocation

Explanation: ERROR—An immediate mode statement attempted to allocate storage, for example, to create a new variable.

User Action: None. You cannot create new storage in immediate mode.

IMMNOTANS, immediate mode not valid when ANSI

Explanation: ERROR—An immediate mode statement was typed when in ANSI mode.

User Action: None.

IMPCNTNOT, implied continuation not allowed

Explanation: ERROR—The program contains an implied continuation line after a statement that does not allow implicit continuation, for example, a DATA statement.

User Action: Use an ampersand (&) to continue the statement.

IMPDECILL, implicit declaration of <name> illegal in immediate mode

Explanation: ERROR—A new variable was named in an immediate mode statement after a STOP, for example, PRINT B after a STOP in a program that has no variable named B.

User Action: None. You cannot create new variables in immediate mode after a STOP statement.

IMPDECNOT implied declaration not allowed for <name> with /EXPLICIT

Explanation: ERROR—A program compiled with the /TYPE=EXPLICIT qualifier contains an implicitly declared variable.

User Action: Declare the variable explicitly or compile the program without the /TYPE=EXPLICIT qualifier.

INACODFOL, inaccessible code follows line <n> statement <m>

Explanation: WARNING—The compiler has detected code that will never be executed, for example, a multistatement line whose first statement is a GOTO, EXIT, ITERATE, RESUME, or RETURN. (VAX BASIC only)

User Action: Review the program to determine if the code should be executed. If you determine the code should be executed, then you should revise the program flow logic accordingly. Otherwise, the code is unnecessary and you may want to remove it. In the case of the GOTO, EXIT, ITERATE, RESUME, or RETURN statements, make sure that these statements are the only statements on the line, or the last statement on a multistatement line.

INCDIRSYN, INCLUDE directive syntax error

Explanation: ERROR—A %INCLUDE directive either is not followed by a quoted string or incorrectly uses the %FROM %CDD or %FROM %LIBRARY clause.

User Action: Supply either a quoted string or the correct syntax for the %FROM %CDD or %FROM %LIBRARY clause.

INCFUNUSA, inconsistent function usage for function <name>

Explanation: ERROR—The parameter list in a DEF function invocation contains a string where the function expected a number, or vice versa. This message is issued only when the invocation occurs before the DEF statement in the program.

User Action: Supply a correct parameter in the function invocation or correct the parameter list in the DEF.

INCRMSERR, INCLUDE directive RMS error number <number>

Explanation: ERROR—A %INCLUDE directive caused an RMS error when accessing the specified file.

User Action: Take action based on the reported RMS error number.

INCSUBUSE, inconsistent subscript use for <array-name>

Explanation: ERROR—The number of subscripts in an array reference does not match the number of subscripts specified when the array was created.

User Action: Specify the same number of subscripts.

INIOUTRAN, initial value must be within the specified range

Explanation: ERROR—The specified initial value is not within the range specified in the RANGE clause.

User Action: Change either the initial value or the range values so that the initial value falls within the range.

INPPROMUS, input prompt must be a string constant

Explanation: ERROR—An INPUT, LINPUT, or INPUT LINE statement list contains a numeric constant immediately following the statement.

User Action: Remove the numeric constant. You can specify only a string constant immediately after an INPUT, LINPUT, or INPUT LINE statement.

INSERTB, assuming <keyword> before <keyword>

Explanation: ERROR—The program contains a syntax error. HP BASIC assumes a keyword is missing and continues compilation under that assumption so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

INSERTM, assuming <keyword> to match <keyword>

Explanation: ERROR—The program contains a syntax error. HP BASIC assumes a keyword is misspelled and continues compilation under that assumption so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

INSSPADYN, insufficient space for MAP DYNAMIC variable in MAP <name>

Explanation: ERROR—A variable named in a MAP DYNAMIC statement is larger than the MAP, for example, a GFLOAT variable in a MAP that is only four bytes long.

User Action: Increase the size of the MAP so that it is large enough to hold the largest member.

INTCODERR, an internal coding error has been detected. Submit an SPR.

Explanation: ERROR—An error has been detected in the HP BASIC compiler.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces this error.

INTCONEXC, integer constant exceeds machine integer size

Explanation: ERROR—The value specified in a DECLARE CONSTANT statement exceeds the largest allowable value for an integer.

User Action: Supply a value in the valid range.

INTCONREQ, integer constant required

Explanation: ERROR—The program contains a noninteger named constant in a context that requires an integer. For example:

```
DIM A ('123'D)
```

User Action: Supply an integer constant.

INTDATTYP, integer data type not supported in ANSI

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains an integer variable or array.

User Action: Remove the integer variable or array.

INTERR, integer error or overflow

Explanation: WARNING—The program contains an integer expression whose value is outside the valid range.

User Action: Modify the expression so that its value is within the allowable range or use an integer data type that can contain all possible values for the expression.

INTERRDES, please submit an SPR —- internal error in comment processing

Explanation: WARNING—An error has been detected while processing a comment in the HP BASIC compiler.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces the error.

INTERRSCA, please submit an SPR —- internal error in SCA support

Explanation: ERROR—An error has been detected in the SCA support in the HP BASIC compiler. If you recompile your program without the /ANALYSIS_DATA qualifier, this error should no longer appear.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces the error.

INVCHNNUM, invalid channel number, must be greater than zero

Explanation: ERROR—A channel number less than or equal to zero was specified.

User Action: Change the channel number to be greater than zero.

INVCONREQ, invalid conversion requested

Explanation: ERROR—The program contains a reference to the REAL or INTEGER functions and the argument is an entire array, GROUP, RECORD, or RFA expression.

User Action: Remove the invalid argument. The argument to these functions must be a numeric expression.

INVINTTYP, invalid integer type

Explanation: ERROR—A reference to the INTEGER function contains an invalid data-type keyword, for example, A = INTEGER(A, SINGLE).

User Action: Change the invalid data-type keyword. The INTEGER function returns only BYTE, WORD, LONG, or QUAD values.

INVLOGNAM, invalid logical name

Explanation: ERROR—The argument to the ASSIGN compiler command specified a logical name length of less than 1 or greater than 63.

User Action: Supply a valid logical name.

INVPRISPE, invalid priority specification, expecting < or >

Explanation: ERROR—On the SET INPUT PRIORITY statement, you specified a character other than < or > to indicate the relative priorities of the two transformation numbers.

User Action: Specify the priority relationship with less than < (lower priority) or greater than > (higher priority).

INVREATYP, invalid real type

Explanation: ERROR—A reference to the REAL function contains an invalid data-type keyword, for example, A = REAL(A, LONG).

User Action: Change the invalid data-type keyword. The REAL function returns only SINGLE, DOUBLE, GFLOAT, SFLOAT, TFLOAT, XFLOAT, or HFLOAT values.

INVSUBTYP, <data-type> is not a subtype of <data-type>

Explanation: ERROR—The program contains an invalid declaration containing contradictory type declarations, for example, DECLARE REAL BYTE.

User Action: Supply a valid declaration. Use only valid integer subtypes for INTEGER and only valid floating-point subtypes for REAL.

IS_NOTDYN, <name> is not a DYNAMIC MAP variable of MAP <name>

Explanation: ERROR—A REMAP statement names a variable that was not named in the MAP DYNAMIC statement for that MAP.

User Action: Remove the variable from the REMAP statement or name the variable in the MAP DYNAMIC statement for that map.

ITEMUSAPP, ITERATE must appear within a loop

Explanation: ERROR—The program contains an ITERATE statement that is not within a FOR-NEXT, WHILE, or UNTIL loop.

User Action: Remove the ITERATE statement, or surround it with a loop.

JMPBADBLO, jump to line number <line number> is into a controlled block

Explanation: ERROR—The program attempts to transfer control to a WHEN block or associated handler.

User Action: Change the program logic so that it does not transfer control to a WHEN block or associated handler.

JMPBADLAB, jump to label: <label> is into a block

Explanation: ERROR—The program attempted to transfer control into a FOR-NEXT, WHILE, UNTIL, IF, or SELECT-CASE block.

User Action: Change the program logic so that it does not transfer control into a block.

JMPBADLIN, jump to line number <number> is into a block

Explanation: INFORMATION—The program transfers control to a line number within a FOR-NEXT, WHILE, UNTIL, IF, or SELECT-CASE block.

User Action: This is an informational message. However, it is bad programming practice to transfer control into a block.

JMPINTDEF, jump into DEF

Explanation: ERROR—The program attempts to transfer control into a DEF block.

User Action: Change the control statement; you cannot transfer control into a DEF block except by invoking the function.

JMPOUTDEF, jump out of DEF

Explanation: ERROR—The program attempts to transfer control out of a DEF block.

User Action: Change the control statement. Use an EXIT DEF, FNEXIT, FNEND, or END DEF statement to transfer control out of a DEF block.

JMPOUTHAN, jump out of HANDLER

Explanation: ERROR—The program attempts to transfer control out of an error handler.

User Action: Change the control statement. Use an EXIT HANDLER, RETRY, or CONTINUE statement to transfer control out of an error handler.

JMPOUTPRO, jump out of program unit

Explanation: ERROR—In a source file containing more than one program module, a statement attempts to transfer control from one module into another.

User Action: Change the statement that attempts to transfer control; you cannot transfer control into a different program module.

JMPUNRLIN, jump to unreferenceable line number <number>

Explanation: ERROR—A RESUME, GOSUB, or GOTO statement attempts to transfer control to a CASE statement.

User Action: Label or number the SELECT statement and transfer control to the beginning of the SELECT-CASE block.

KEYCANNOT, key <name> in MAP <map-name> cannot be a dynamic variable

Explanation: ERROR—A KEY clause in an OPEN statement specifies a variable declared as dynamic in a MAP DYNAMIC statement.

User Action: Specify a static variable in the KEY clause; that is, declare the variable in a MAP statement, not a MAP DYNAMIC statement.

KEYIS_NEE, key is needed for indexed files

Explanation: ERROR—The program attempts to open an indexed file for output, and the PRIMARY KEY clause is missing.

User Action: Supply a PRIMARY KEY clause.

KEYMUSBE, key must be either word, longword, quadword, string, decimal, record or group

Explanation: ERROR—A FIND or GET statement on an indexed file contains a key specification that is not a WORD, LONG, QUAD, STRING, DECIMAL, or an 8-byte RECORD or GROUP expression.

User Action: Change the key specification to be a WORD, LONG, QUAD, STRING, DECIMAL, or an 8-byte RECORD or GROUP expression.

KEYMUSTBE, key, <vbl-name> in map <map-name> must be either word, longword, quadword, string, decimal, record or group

Explanation: ERROR—An OPEN statement contains a key specification that is not an unsubscripted WORD, LONG, QUAD, STRING, DECIMAL, or an 8-byte RECORD or GROUP variable.

User Action: Change the key specification to be an unsubscripted WORD, LONG, QUAD, STRING, DECIMAL, or an 8-byte RECORD or GROUP variable.

KEYNOTMAP, KEY <vbl-name> is not an unsubscripted variable in MAP <name>

Explanation: ERROR—An indexed file OPEN statement specifies a KEY variable that does not appear in a MAP statement.

User Action: Place the KEY variable in the MAP referenced by the OPEN statement's MAP clause.

KEYREQMAP, KEY clauses require a MAP clause

Explanation: ERROR—An OPEN statement specifies KEY clauses without specifying a MAP clause.

User Action: Supply a MAP clause to define the position of the keys in the record buffer.

KEYSEGMUS, key segment <name> in map <map-name> must be a string key

Explanation: ERROR—An OPEN statement specifies a segmented key containing a numeric variable. For example:

```
OPEN "INDEX.DAT" AS FILE #1, ORGANIZATION INDEXED,  &
    PRIMARY KEY (A$, B$, C%), MAP ABC
```

User Action: Specify only string variables in segmented keys.

KEYSINC, <keyword> keyword inconsistent with <keyword>

Explanation: ERROR—An OPEN statement contains contradictory record format specifications, for example, both FIXED and VARIABLE.

User Action: Specify only one record format.

KEYTOOLON, KEY <name> in MAP <name> is too long (max is 255)

Explanation: ERROR—A KEY variable is longer than 255 characters.

User Action: Reduce the length of the KEY variable. The maximum key length is 255 characters.

KEYWORINC, keyword inconsistent with <OPEN clause> clause

Explanation: ERROR—An OPEN statement contains an ALLOW, ACCESS, or RECORDTYPE clause whose keyword argument is invalid, for example, ACCESS FORTRAN.

User Action: Change the clause argument to a valid keyword for that clause.

LABNOTDEF, label <label> not defined

Explanation: ERROR—The program tries to transfer control to a nonexistent label.

User Action: Define the label before transferring control to it.

LABNOTLAB, label <name> does not label an active block statement

Explanation: ERROR—An EXIT statement in a loop, if-then-else, or SELECT-CASE block specifies a label that does not refer to that block.

User Action: Change the program so that the label actually refers to the block in which the EXIT statement occurs.

LABNOTLOO, label <name> does not label an active loop statement

Explanation: ERROR—In a loop, an EXIT or ITERATE statement specifies a label that does not refer to that loop.

User Action: Change the program so that the label actually refers to the loop in which the EXIT or ITERATE statement occurs.

LANFEADEC, language feature is declining

Explanation: INFORMATION—The program contains a language feature that is not recommended for new program development, for example, the FIELD statement. This error is reported only when the /FLAG=DECLINING qualifier is in effect.

User Action: Use: 1) MAP, MAP DYNAMIC and REMAP statements instead of FIELD, 2) EDIT$ rather than CVT$$, and 3) overlaid MAPs rather than CVTxx functions.

LANFEAINC, language feature incompatible with BASIC-PLUS-2

Explanation: INFORMATION—The program contains syntax that results in different behavior under HP BASIC and PDP–11 BASIC-PLUS-2, for example, opening a terminal–format file. This error is reported only when the /FLAG=BP2COMPATIBILITY qualifier is in effect.

User Action: None.

LANFEAINH, language feature inhibits optimization

Explanation: INFORMATION—A program compiled with the /NOSETUP qualifier contains a language feature that requires /SETUP, for example, the RESUME statement. The compilation continues with /SETUP in effect.

User Action: None. The program must be compiled with /SETUP in effect for the language feature to work.

LANFEANOT, language feature not available in BASIC-PLUS-2

Explanation: INFORMATION—The program contains a language element that is not supported in BASIC-PLUS-2, for example, RECORD statements. This error is reported only when the /FLAG=BP2COMPATIBILITY qualifier is in effect.

User Action: If the program must run under both HP BASIC and PDP–11 BASIC-PLUS-2, you must remove the incompatible language feature.

LANFEAOPE, language feature not available in BASIC

Explanation: ERROR—The program contains a PRINT statement with a RECORD clause. BASIC does not support the RECORD clause in PRINT statements.

User Action: Remove the RECORD clause.

LEFBOUSPE, left boundary must be less than the right boundary

Explanation: ERROR—In a statement that specifies a viewport or windowsize, you specified a left boundary that is greater than or equal to the corresponding right boundary.

User Action: Correct the left boundary so that it is less than the right boundary.

LENDYNSTR, string length not allowed on dynamic string <name>

Explanation: ERROR—The program contains a dynamic string variable declaration that specifies a string length.

User Action: Length specifications are allowed only for fixed-length strings; remove the length specification from the dynamic string, or allocate the string in a MAP or COMMON.

LENNUMFIL, string length not allowed on numeric FILL

Explanation: ERROR—The program contains a numeric FILL item that specifies a length.

User Action: Remove the length specification from the numeric FILL item.

LETDIRSYN, LET directive syntax error

Explanation: ERROR—A %LET directive contains a syntax error, for example, an invalid lexical identifier.

User Action: Use the correct syntax for the %LET directive.

LETKEYREQ, LET keyword required in ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains an assignment statement that omits the LET keyword.

User Action: Supply a LET keyword.

LEXDIRSYN, lexical directive syntax error

Explanation: ERROR—A syntax error was detected in a lexical directive.

User Action: Correct the syntax of the lexicial directive.

LEXIDEMUS, lexical identifier must be declared before reference

Explanation: ERROR—You reference a lexical identifier before you declare it.

User Action: Declare the lexicial identifier before you reference it.

LINNOTALL, line numbers not allowed, use the EDIT command

Explanation: ERROR—An EDIT command with a line number has been found in a program without line numbers.

User Action: Use the EDIT command without specifying a line number to invoke a text editor.

LINNUMERR, illegal line number

Explanation: ERROR—The program contains a line number that is outside the valid range or is not a valid integer (note that the percent sign (%) suffix is not valid for line numbers).

User Action: Specify only integer line numbers in the range 1 to 32,767, inclusive.

LINNUMINC, line number may not appear in INCLUDE directive file

Explanation: ERROR—The file specified in a %INCLUDE compiler directive contains a line number.

User Action: Remove the line number from the file.

LINNUMUND, line number <n> undefined due to conditional compilation

Explanation: ERROR—The program references a line number that does not appear in the object code as a result of the branch taken in a %IF-%THEN-%ELSE-%END-%IF directive.

User Action: Change the %IF-%THEN-%ELSE-%END-%IF directive or remove the line number reference.

LINOUTORDE, Line numbers are out of order

Explanation: ERROR—The line numbers in an Alpha BASIC program are not in ascending order.

User Action: Reorder the line numbers and/or statements in your program so that the line numbers are in ascending order.

LINREQTWO, LINES output requires at least 2 X,Y points

Explanation: ERROR—A LINE graphic output statement specifies less than 2 points.

User Action: Specify a minimum of 2 points in the LINE graphic output statement.

LNPNOTBP2, programs without line numbers are not allowed in BASIC-PLUS-2

Explanation: INFORMATION—BASIC-PLUS-2 does not support programs without line numbers. This error is reported only when the /FLAG = BP2COMPATIBILITY qualifier is in effect.

User Action: Add a line number to the first line of the program.

LOGOPENON, logical operation on noninteger quantity

Explanation: ERROR—The program contains a logical operation performed on strings or real numbers.

User Action: Change the logical operands to integers.

LOOINDMUS, loop control variable must be a numeric variable

Explanation: ERROR—A FOR statement specifies a string variable as the loop control variable.

User Action: Specify a numeric variable. You can use only numeric variables as loop control variables.

LOOINIMUS, loop initial value must be a numeric expression

Explanation: ERROR—A FOR statement attempts to assign a string expression as the loop control variable's initial value.

User Action: Remove the string expression. You can assign only numeric values as the loop's initial value.

LOOLIMMUS, loop limit must be numeric

Explanation: ERROR—A FOR statement attempts to assign a string expression as the loop control variable's limiting value.

User Action: Remove the string expression. You can assign only numeric values as the loop control variable's limiting value.

LOOWILNEV, loop will never execute

Explanation: WARNING—The program contains a FOR/NEXT loop that is not executable, for example, FOR I% = 1% TO 0%. Compilation continues, but the loop is ignored.

User Action: Change the loop parameters or insert an appropriate STEP clause.

LOWLSSUP, lower bound must be less than upper bound

Explanation: ERROR—The lower bound specified in the array is greater than the upper bound.

User Action: Correct the bounds.

LOWNOTVIR, lower bound not permitted with virtual arrays

Explanation: ERROR—Lower bounds of virtual arrays must be zero.

User Action: Correct the lower bounds to be zero.

LOWNOTZERO, lower bound must be zero

Explanation: ERROR—The lower bound of the array must be zero.

User Action: Correct the lower bound to be zero.

LOWRANVAL, range lower value must be less than upper value

Explanation: ERROR—In the RANGE clause, the first value is greater than the second value.

User Action: Change the range clause so that the first value is less than the second value.

LRSETNOT, <keyword> is not allowed with MID

Explanation: ERROR—The LSET and RSET keywords are not allowed with MID.

User Action: Change the LSET or RSET keyword to LET.

MACNOTDEF, macro is not defined

Explanation: ERROR—The macro identifier used in this %UNDEFINE directive was not previously defined by a %DEFINE directive.

User Action: Verify that the identifier has been previously defined with the %DEFINE directive. Verify that the %DEFINE and %UNDEFINE macro IDs match.

MAPDYNNOT, MAP DYNAMIC <map-name> may not be larger than 32,767 bytes

Explanation: ERROR—A MAP DYNAMIC statement references a map that is greater than 32,767 bytes in size.

User Action: Reduce the size of the map, as defined in the MAP statement, or MAP statements, to 32,767 bytes or less.

MAPDYNREQ, MAP DYNAMIC <name> requires corresponding static MAP

Explanation: ERROR—The program contains a MAP DYNAMIC statement whose MAP name does not appear in a MAP statement.

User Action: Provide a MAP with the same name as the MAP DYNAMIC name.

MAPNOTDEF, MAP <name> used in OPEN not defined

Explanation: ERROR—An OPEN statement's MAP clause references a nonexistent MAP.

User Action: Define the MAP referenced by the MAP clause, or remove the MAP clause.

MAPTOOLAR, MAP too large in OPEN

Explanation: FATAL—The size of the MAP referenced in an OPEN statement is greater than 32,767 bytes.

User Action: Reduce the size of the MAP.

MAPVARALI, variable <name> not aligned in multiple references in MAP <name>

Explanation: ERROR—More than one overlaid MAP contains the same variable, but the variable's position differs in the MAPs.

User Action: The same variable can appear in multiple overlaid MAPs, but the variable must occupy the same position in the PSECT; make sure that the variable appears in the same position in the MAPs.

MAPVARREF, MAP variable <name> referenced before declaration

Explanation: INFORMATION—A reference to a MAP variable occurs before the MAP statement.

User Action: Make sure that the MAP statement precedes any references to variables in the MAP.

MATDIMERR, matrix dimension error

Explanation: ERROR—The program either:

- Contains a MAT IDN, MAT TRN, or MAT INV performed on a one-dimensional array
- Performs a matrix operation that requires identical subscripts in the operand arrays and those arrays have different subscripts

User Action: Dimension the arrays to the proper number of subscripts.

MATLOWBOU, matrix must have lower bound 0 and upper bound 4

Explanation: ERROR—The specified transformation matrix either has a lower bound other than 0 or an upper bound other than 4.

User Action: Declare the matrix such that both dimensions have a lower bound of 0 and an upper bound of 4.

MATMUL2OP, MAT multiply of 2 4X4 matrices required

Explanation: ERROR—You specified the wrong dimensions in a matrix in the MAT multiply statement or a WITH clause on the DRAW statement, or you specified a nonmultiplication operation in a multiple operation MAT arithmetic statement. For example: MAT A=B*C+D. A two-dimensional matrix with lower bounds 0 and upper bounds 4 in both dimensions is required.

User Action: Declare the matrix to be a two-dimensional matrix with lower bounds 0 and upper bounds 4 in both dimensions.

MATONEOR2, MAT statements require one or two dimensions

Explanation: ERROR—A MAT statement references an array of more than two dimensions.

User Action: Remove the array reference. MAT statements are valid only on arrays of one or two dimensions.

MAXCONCOM, maximum conditional compilation depth exceeded

Explanation: FATAL—Too many nested %IF-%THEN-%ELSE-%END-%IF directives are contained in the program.

User Action: Reduce the number of nested %IF-%THEN-%ELSE-%END-%IF directives. You can nest up to eight such constructs.

MAXDIMEXC, maximum number of dimensions exceeded. Maximum is 32

Explanation: ERROR—An array declaration specifies more than the allowed number of dimensions.

User Action: Reduce the number of dimensions to 32 or less.

MAXKEYSEG, maximum of 8 key segments exceeded

Explanation: ERROR—An OPEN statement specifies a segmented key with more than eight segments.

User Action: Reduce the number of segments in the KEY clause to eight or less.

MAXPAREXC, maximum parameters exceeded for <name>. Maximum is <number>

Explanation: ERROR—The program attempts to declare a DEF with more than eight parameters or a subprogram with more than 255 parameters.

User Action: Reduce the number of parameters; DEFs allow up to eight parameters and subprograms allow up to 255 parameters.

MAXPAREXP, no more than <number> parameter(s) expected for <sub–func–name>

Explanation: ERROR—An external SUB or FUNCTION was called with more parameters than were specified in the EXTERNAL statement, including both OPTIONAL and non-OPTIONAL parameters.

User Action: Reduce the number of parameters in the call.

MERGE, merged <item> and <item>

Explanation: ERROR—The program contains a syntax error. HP BASIC assumes that there is an incorrect space, for example, PR INT. Compilation continues so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

MINPAREXP, at least <number> parameter(s) expected for <sub–func–name>

Explanation: ERROR—An external SUB or FUNCTION was called with fewer parameters than were specified as non-OPTIONAL parameters in the EXTERNAL statement.

User Action: Increase the number of parameters in the call so that the number of parameters is equal to or greater than the number of non-OPTIONAL parameters.

MISENDIF, missing END IF directive before end of program unit

Explanation: ERROR—A %IF directive crosses a program module boundary.

User Action: Terminate the %IF with a %END %IF before beginning a new source module.

MISENDFOR, missing END <block> for <block> at line <n> statement <m>

Explanation: ERROR—The program contains a SELECT, IF, or DEF without a matching END statement.

User Action: Supply a matching END statement.

MISMATEND, mismatched END, expected <block>

Explanation: ERROR—The program contains an incorrect END statement, for example, an END RECORD statement instead of an END GROUP statement.

User Action: Supply the correct type of END statement.

MISMATFOR, missing NEXT for <item> at line <n> statement <m>

Explanation: ERROR—The program contains a FOR, WHILE, or UNTIL without a matching NEXT.

User Action: Supply the matching NEXT statement.

MODNOTFND, module <mod-name> not found in text library <text-lib-name>

Explanation: ERROR—The module name you specified in a %INCLUDE directive was not found in the text library you specified.

User Action: Place the module name in the specified text library.

MULCHRARR, multiple character array name not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains an array whose name contains more than one character.

User Action: Reduce the length of the name to a single character.

MULCHRDEF, multiple character DEF name not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains a DEF whose name contains more than one character.

User Action: Reduce the length of the name to a single character.

MULDEFLEX, multiple definition of lexical identifier is illegal

Explanation: ERROR—A lexical constant is named in more than one %LET directive.

User Action: Declare the lexical constant only once with %LET.

MULHANSPE, multiple handlers specified for WHEN block

Explanation: ERROR—A WHEN block specifies both an attached and detached error handler.

User Action: Change the WHEN block to specify either an attached or detached error handler.

MULMAIPROG, multiple main program units are illegal

Explanation: ERROR—More than one main program unit has been detected in a single source file.

User Action: Modify your source file so that all HP BASIC statements are contained within a single main program or within a subprogram.

MULNOTBP2, multiple program units per module not BASIC-PLUS-2 compatible

Explanation: INFORMATION—A program compiled with the /FLAG=BP2COMPATIBILITY qualifier contains more than one program unit. BASIC-PLUS-2 does not allow more than one program unit in a single file.

User Action: Separate the program into individual program units and compile the units separately.

MULOPTBAS, multiple OPTION BASE statements not ANSI

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains more than one OPTION BASE statement.

User Action: Specify the OPTION BASE statement only once per program.

MULPRONOT, multiple program units per module not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains more than one program unit.

User Action: Rewrite the program converting the subprograms to subroutines.

MULSTAPER, multiple statements per line not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains more than one statement on a line.

User Action: Change the program so that each statement has its own line number.

MULTDEF, multiple definition of <name>

Explanation: WARNING—A variable is declared in more than one declarative statement.

User Action: Make sure that the variable is declared only once.

NAMNOTREC, name <name> is not of a RECORD component

Explanation: ERROR—A RECORD component reference uses an invalid record name, for example, A::B when A is not a RECORD name.

User Action: Change the erroneous reference.

NAMTOOLON, name is too long, changed to <name>

Explanation: WARNING—A variable or array name is longer than 31 characters. HP BASIC truncates the name to 31 characters and continues compilation so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Reduce the length of the variable name to 31 or fewer characters.

NEGFILSTR, negative FILL or string length

Explanation: ERROR—The program contains a negative FILL specification or string length.

User Action: Change the FILL specification or string length to a positive number.

NESFORLOO, nested FOR loops with same control variable <name>

Explanation: ERROR—The program contains nested FOR/NEXT loops that use the same index variable.

User Action: Change the index variable for all but one of the loops.

NOBASFRAM, no BASIC frame on stack

Explanation: ERROR—HP BASIC could not find a valid stack frame. This could be caused by running a program with /CHECK=NOBOUNDS or by a non-BASIC subprogram.

User Action: Debug the program before running with /CHECK=NOBOUNDS or check the logic of the non-BASIC subprogram.

NODESCALL, no descriptor allocated for array <name>

Explanation: ERROR—An immediate mode statement required an array descriptor, but it was not available. HP BASIC allocates array descriptors only if the program code requires it.

User Action: None.

NODIAGFILE, unsaved changes, no diagnostics file produced

Explanation: WARNING—The program in memory contains changes that have not been saved; therefore, no diagnostics file will be produced from this compilation.

User Action: SAVE or REPLACE the file.

NOEDIT, no change made

Explanation: WARNING—The search string in an EDIT command was not located in the text.

User Action: Enter valid search string.

NOFILEALL, a file specification is not allowed with the REPLACE command

Explanation: ERROR—The REPLACE command does not allow the use of a file specification.

User Action: Use either the SAVE command with a file specification or the REPLACE command without one.

NOFRAME, compiled procedure is currently not active

Explanation: WARNING—A program containing multiple compilation units has been stopped while running in the environment due to a STOP statement or a Ctrl/C entered by the user. The NOFRAME error indicates that the routine executing when the program stopped is not the last compilation unit in the source file. You can examine or modify a variable in immediate mode only if the current routine is the last compilation unit in the source file.

User Action: If you do not intend to debug your program in immediate mode, no action is required. If you do intend to debug your program in immediate mode, make the routine that you want to debug the last compilation unit in the source file. The symbols for the last compilation unit are always available. Alternatively, use the OpenVMS Debugger to debug your program.

NOHANSPE, no handler specified for WHEN block

Explanation: ERROR—A WHEN block has been found that does not specify an error handler.

User Action: Specify an error handler for the WHEN block.

NOLINENUM, missing line number on first line

Explanation: WARNING—There is no line number on the first line of the program.

User Action: Add a line number to the first line of the program or remove all line numbers from the program.

NOLNROOM, out of memory for line numbers

Explanation: ERROR—The program contains more line-numbered statements than HP BASIC allows.

User Action: Change the program so that it uses multistatement lines instead of having each statement on its own line or split the program into one or more program units in separate files.

NOMAPNAME, MAP statement requires map name

Explanation: ERROR—A MAP statement does not specify a map name.

User Action: Specify a name for the MAP.

NOSCAFILE, unsaved changes, no analysis file produced

Explanation: WARNING—The program in memory contains changes that have not been saved; therefore, no data analysis file will be produced from this compilation.

User Action: SAVE or REPLACE the file.

NOSRCLINE, unsaved changes, no source line debugging available

Explanation: WARNING—The program in memory contains changes that have not been saved; therefore, no source line debugging will be available from this compilation.

User Action: SAVE or REPLACE the file.

NOSUCHMAP, no such MAP area <name>

Explanation: ERROR—A REMAP statement names a nonexistent MAP area.

User Action: Supply a MAP before executing the REMAP statement.

NOTIMP, not implemented in this version

Explanation: ERROR—The program attempted to use a feature that does not exist in this version of HP BASIC.

User Action: Examine your program and remove the nonimplemented feature.

NOTMACROOM, out of memory for macro definitions

Explanation: ERROR—The limit on the number of macro definitions has been exceeded.

User Action: Reduce the number of %DEFINE definitions, or undefine some previously defined macros with the %UNDEFINE directive.

NOTPASSBY, <item> may not be passed BY <mechanism>

Explanation: ERROR—The program specifies an incorrect passing mechanism for a parameter's data type, or an invalid parameter. For example, you cannot pass an entire array BY VALUE, nor can you pass a label as a parameter.

User Action: Specify a valid parameter or passing mechanism.

NOTRANS, no main program

Explanation: WARNING—When a RUN command was typed, only subroutines or functions were available. HP BASIC requires a main program to receive the transfer of control.

User Action: Supply a main program.

NOTRECVBY, <item> may not be received by <mechanism>

Explanation: ERROR—A subprogram specifies an invalid parameter or an incorrect passing mechanism for a parameter's data type. For example, you cannot receive an entire array
BY VALUE.

User Action: Specify a valid parameter or passing mechanism.

NOTXTROOM, out of memory for statement text

Explanation: ERROR—The program contains more text than HP BASIC allows.

User Action: Split the program into one or more program units.

NOVALUE, <text> keyword requires a value

Explanation: ERROR—A keyword command was typed without a value.

User Action: Supply a valid keyword value.

NUMARREXP, numeric array expected

Explanation: ERROR—A CHANGE statement does not specify a numeric array.

User Action: Supply a numeric array in the CHANGE statement.

NUMCONREQ, numeric constant required

Explanation: ERROR—The program contains a string in a context that requires a numeric constant. For example:

```
DECLARE INTEGER CONTANT A = "ABC"
```

User Action: Supply a numeric constant.

NUMIS_NEE, numeric expression is required

Explanation: ERROR—The program contains a string expression in a context that requires a numeric expression, for example, WHILE A$.

User Action: Supply a numeric expression.

NUMVARREQ, numeric variable required

Explanation: ERROR—A nonnumeric variable was found with a numeric data type.

User Action: Specify a numeric variable.

OBJFAIL, failure in loading object file

Explanation: FATAL—In the VAX BASIC Environment, either an attempt was made to load a non-BASIC object module, or the compiler could not find the object file referenced by a CALL statement or EXTERNAL FUNCTION reference.

User Action: If the object file resides in the Common Run-Time Library, you must link the program at DCL level. If the object file is in a user-supplied library, use the DCL LIBRARY command to install the missing object module. You can load only VAX BASIC object modules.

ONENOTWHE, ON ERROR not allowed in WHEN block or handler

Explanation: ERROR—An ON ERROR statement has been found in a WHEN block or an associated error handler.

User Action: Remove the ON ERROR statement from the WHEN block or associated error handler.

OPEEXPNOT, operator expected, not found

Explanation: ERROR—A compiler directive contains an invalid lexical expression that has a right parenthesis immediately followed by a lexical identifier.

User Action: Correct the lexical expression.

OPEMUSFOL, operator must follow right parenthesis

Explanation: ERROR—The program contains an incorrect lexical expression.

User Action: Correct the lexical expression.

OPENIN, error opening <file-name> as input

Explanation: ERROR—An error was detected in attempting to open a file for input.

User Action: Make sure the file specification is correct.

OPENOUT, error opening <file-name> as output

Explanation: ERROR—An error was detected in attempting to open a file for output.

User Action: Supply a valid file specification, or take corrective action based on the associated message.

OPNCLAVAL, OPEN clause <clause> value greater than <number>

Explanation: ERROR—An OPEN statement contains a RECORDSIZE, FILESIZE, EXTENDSIZE, WINDOWSIZE, BLOCKSIZE, BUCKETSIZE, or BUFFER clause whose argument is too large.

User Action: Supply a smaller value for the argument.

OPNDUPCLA, duplicate OPEN clause

Explanation: WARNING—An OPEN statement contains more than one clause of the same type.

User Action: Remove one of the clauses.

OPNILLCLA, <clause> is an unsupported OPEN clause

Explanation: ERROR—An OPEN statement specifies invalid attributes for the file, for example, CLUSTERSIZE on OpenVMS systems, or uses the keyword COMMON in an I/O clause.

User Action: Substitute valid attributes for the file or remove the COMMON keyword.

OPNINCCLA, <keyword> keyword is inconsistent with file organization

Explanation: ERROR—An OPEN statement contains a clause that is not appropriate for the specified file organization, for example, opening a relative file with the ACCESS APPEND clause.

User Action: Remove the inconsistent clause.

OPTBASMUS, OPTION BASE must be before array declarations

Explanation: ERROR—A program compiled with the /ANSI_STANDARD qualifier contains an OPTION BASE statement that lexically follows an array declaration.

User Action: Move the OPTION BASE statement so that it lexically precedes the array declaration.

OPTCLACON, OPTION clause occurs more than once

Explanation: ERROR—The OPTION statement contains a duplicate clause, for example, specifying the default integer size as both BYTE and LONG.

User Action: Remove one of the clauses.

OPTNOTALL, OPTIONAL not allowed on EXTERNAL PICTURE

Explanation: ERROR—An attempt was made to specify the OPTIONAL keyword on an EXTERNAL PICTURE declaration. This is not allowed because OPTIONAL parameters should be used for calling non-BASIC procedures only.

User Action: Remove the OPTIONAL keyword from the EXTERNAL PICTURE declaration.

OPTOUTSEQ, OPTION statement out of sequence

Explanation: ERROR—The OPTION statement is either: 1) not the first statement in a main program, or 2) not the first statement following the SUB or FUNCTION statement.

User Action: Move the OPTION statement so that it is either the first statement in the main program or the first statement following the SUB or FUNCTION statement in the subprogram.

ORGUNDREQ, ORGANIZATION UNDEFINED requires FOR INPUT clause

Explanation: ERROR—The program opens a file with ORGANIZATION UNDEFINED, but does not specify FOR INPUT.

User Action: Specify FOR INPUT in the OPEN statement. You cannot create a file with an undefined file organization.

OVFCHKSUP, OVERFLOW checking supported only for INTEGER and DECIMAL

Explanation: ERROR—Overflow checking was specified for a floating-point data type in: 1) a compiler command, 2) a qualifier to the DCL BASIC command, or 3) an OPTION statement.

User Action: Specify overflow checking only for INTEGER or DECIMAL data types or both.

OVRNOLINE, <keyword> overrides NOLINE

Explanation: WARNING—The program: 1) was compiled with /NOLINES and 2) uses a keyword that requires line number information. For example, ERL and RESUME with a line number both require that the program be compiled with /LINES.

User Action: None. If you use a keyword that requires line number information, VAX BASIC automatically overrides the /NOLINE qualifier.

PAREXPFOR, <n> parameters expected for <routine>

Explanation: ERROR—The CALL or invocation of a routine specifies a different number of parameters than the number specified when the routine was declared.

User Action: Change the number of parameters to match the number declared.

PARINCPRE, parameter <name> inconsistent with previous declaration or reference

Explanation: ERROR—An external subprogram or DEF function declaration specifies a data type for one of the parameters that is different than the data type the SUB, FUNCTION, or DEF statement specifies.

User Action: Change the specified data type in either the declaration or the SUB, FUNCTION, or DEF statement so that the data types agree.

PARMODCHA, mode for parameter <n> of routine <name> changed to match declaration

Explanation: INFORMATION—The data type specified in a routine invocation does not match that of the routine declaration. HP BASIC issues this message only if the data-type conversion results in a parameter that cannot be modified by the routine that was invoked.

User Action: Make the data-type specifications in the declaration and the invocation match.

PARMODNOT, mode for parameter <n> of routine <name> not as declared

Explanation: ERROR—The CALL or invocation of a routine specifies a string argument for a parameter that was specified as a numeric when the routine was declared, or vice versa.

User Action: Change the string parameter to numeric, or vice versa.

PARNOTENT, parenthesis illegal, entire array required context

Explanation: ERROR—Parenthesis are used to specify an entire array in a context where an entire array is always required.

User Action: Remove the empty parenthesis from the entire array reference.

PARSTRNOT, parameter <n> of <type> structure not as declared

Explanation: ERROR—The actual parameter list in subprogram CALL or an invocation specifies an entire array where the subprogram declaration specified a simple variable or vice versa.

User Action: Change the actual parameter list to match the declared parameter list or vice versa.

PARTYPREQ, parameter type specification required with /EXPLICIT

Explanation: ERROR—In a program compiled with /TYPE=EXPLICIT, no data-type keyword is specified for a parameter.

User Action: Supply a data-type keyword for the parameter. There are no default data types when you compile a program with /TYPE=EXPLICIT.

PASMECDEF, passing mechanism not allowed for DEF

Explanation: ERROR—A DEF function declaration specifies a passing mechanism for a parameter.

User Action: Remove the passing mechanism clause.

PASMECDIS, passing mechanism disagrees with declaration

Explanation: ERROR—The CALL or invocation of a routine specifies a different passing mechanism for a parameter than that specified when the routine was declared.

User Action: Remove the BY clause specified in the CALL or invocation; HP BASIC automatically passes parameters with the passing mechanism specified when the routine was declared.

PASMECNOT, passing mechanism not allowed for <item>

Explanation: ERROR—A program specifies a passing mechanism in a context other than the invocation or declaration of an external subprogram.

User Action: Remove the passing mechanism clause.

PASWITNO, <name> has a passing mechanism specified with no parameter list

Explanation: WARNING—A CALL statement, external function reference, or EXTERNAL statement specifies a BY clause but does not specify a formal parameter list.

User Action: Remove the BY clause or supply a parameter list.

PATNOTREC, path name does not specify a CDD/Repository record

Explanation: ERROR—The %INCLUDE directive contains an invalid path name for a record definition.

User Action: Supply a valid path name for a record definition.

PICWHINOT, exit from PICTURE while not in PICTURE

Explanation: ERROR—An EXIT PICTURE statement was found in a module that is not a PICTURE subprogram.

User Action: Remove the EXIT PICTURE statement.

PLACENODESIGN, placeholders illegal without /DESIGN=PLACEHOLDERS

Explanation: ERROR—A placeholder occurred in the source file and the /DESIGN=PLACEHOLDERS option was not specified.

User Action: Recompile the program and specify the qualifier.

PLACENODOT, repetition of pseudocode placeholders not allowed

Explanation: ERROR—A pseudocode placeholder was syntactically incorrect.

User Action: You should remove the trailing periods following the pseudocode placeholder.

PLACENOEXE, placeholders detected—source cannot be executed

Explanation: ERROR—The source code for a RUN command in immediate mode contained at least one placeholder, therefore it could not be executed.

User Action: You should remove the placeholders from the source code and reissue the command.

PLACENOOBJ, placeholders detected—no object produced

Explanation: INFORMATION—The program contained one or more placeholders and therefore no object module was created.

User Action: Remove the placeholders from the source code.

PLACEUNMAT, unmatched placeholder delimiter

Explanation: ERROR– A placeholder was syntactically incorrect because the number of opening and closing brackets did not match.

User Action: First, make sure that the placeholder does not span multiple source lines. Second, add or remove brackets until they are appropriately paired.

PLACEWRDOT, invalid placeholder repetition

Explanation: ERROR—A list placeholder was syntactically incorrect. Three periods were expected.

User Action: Add trailing periods until there are three periods following the placeholder.

POIREQONE, POINTS output requires at least 1 X,Y point

Explanation: ERROR—You do not specify a point in the POINT graphic output statement.

User Action: Specify a minimum of 1 point in the POINT graphic output statement.

POSGTRTAR, starting position greater than target length

Explanation: ERROR—The starting value in the MID statement is greater than the length of the string.

User Action: Correct the value to be less than or equal to the length of the string.

PRELOGNAM, previous logical name assignment replaced

Explanation: INFORMATION—The specified logical name already existed. The new equivalence name replaces the old one.

User Action: None.

PRICDDERR, prior severe CDD/Repository error

Explanation: ERROR—There have been one or more severe CDD/Repository errors, and this may be the reason for the following errors.

User Action: Recompile the program after correcting the first errors related to CDD/Repository.

PRIUSICLA, PRINT USING clause must be a string expression

Explanation: ERROR—A PRINT USING statement specifies a numeric format string.

User Action: Supply a valid format string.

PRIUSICON, PRINT USING conflicts with RECORD clause

Explanation: ERROR—A PRINT USING statement contains a RECORD clause.

User Action: Remove the RECORD clause or use the PRINT statement instead of PRINT USING.

PROSTRNES, program structures nested too deeply

Explanation: FATAL—The program contains too many nested block constructs, for example, DEF function definitions.

User Action: Reduce the number of nested block constructs.

PROTOOBIG, program too big to compile

Explanation: FATAL—The program is too big.

User Action: Recode the program as two or more modules.

PROWHINOT, exit from PROGRAM while not in a main program

Explanation: ERROR—An EXIT PROGRAM statement was found in a program unit that is not a main program.

User Action: Use the type of EXIT appropriate to the program unit.

QUALERR, unknown qualifier <name>

Explanation: ERROR—An attempt was made to enter an invalid qualifier to a SET, LOCK, or COMPILE command.

User Action: Enter the SET, LOCK, or COMPILE command with the correct qualifier.

RADNOTSUP, radix not supported

Explanation: ERROR—A literal constant specifies a radix. For example, in the following DECLARE statement, H is an invalid radix specifier:

```
10      DECLARE LONG CONSTANT A = H"111"
```

User Action: Specify a valid radix. See the *HP BASIC for OpenVMS Reference Manual* for a list of the radices HP BASIC allows.

REAACCINC, READ access inconsistent with FOR OUTPUT

Explanation: ERROR—An OPEN statement specifies FOR OUTPUT and ACCESS READ.

User Action: FOR OUTPUT specifies that a new file is created; ACCESS READ specifies that the program can only read the file. If you want to create a new file, remove the ACCESS READ clause; if you want read-only access to a file, specify FOR INPUT.

READERR, error reading <file-name>

Explanation: ERROR—An error was detected in attempting to read a file.

User Action: Supply a valid file specification or take corrective action based on the associated message.

REAWITDAT, READ without DATA statement

Explanation: ERROR—The program contains a READ statement and there are no DATA statements.

User Action: Supply a DATA statement or remove the READ statement.

RECENTARR, RECORD entire array must not have subfields specified

Explanation: ERROR—A RECORD component reference specifies an array before the end of the component path, for example, A::B(,)::C.

User Action: Remove the erroneous reference.

RECFILTOO, <field-name> from CDD/Repository has FILL too large

Explanation: ERROR—The total size of a CDD/Repository record is greater than 65,535 bytes.

User Action: Reduce the size of the record.

RECKEYQAD, entire RECORD or GROUP must be 8 bytes in length

Explanation: ERROR—The user attempts to specify an entire RECORD or GROUP name in a key value field on a GET or FIND statement and the size of the structure does not match the size of the QUADWORD.

User Action: When specifying a quadword key, use an 8-byte RECORD or GROUP; otherwise, specify the name of an elementary item in the RECORD or GROUP.

RECNOTBY, record may not be passed BY <mechanism>

Explanation: ERROR—The program attempts to pass a record to a subprogram using either the BY VALUE or BY DESC parameter-passing mechanism.

User Action: Remove the passing mechanism, or specify BY REF. HP BASIC programs can pass records only by reference.

RECNOTDEF, record type <name> not defined

Explanation: ERROR—The program declares an instance of a user data type, but this type was not defined in the program module.

User Action: Define the data type with a RECORD statement.

RECOVEMAP, RECORDSIZE overflows MAP

Explanation: ERROR—An OPEN statement contains both a RECORDSIZE clause and a MAP clause, and the RECORDSIZE clause is larger than the MAP.

User Action: Make the RECORDSIZE the same as the MAP size.

RECRECDEF, recursive RECORD definition of type <name>

Explanation: ERROR—The program contains two or more RECORD statements that reference each other.

User Action: Change the program so that the RECORD statements do not point at each other.

RECTOBIGL, record too big from module <mod-name> in text library <text-lib-name>

Explanation: ERROR—the text library module specified in a %INCLUDE directive contains a record longer than 255 bytes.

User Action: Extract the module from the text library, edit it to remove any records longer than 255 bytes, and replace the module in the text library.

RECTOOBIG, record too big from INCLUDE directive file

Explanation: ERROR—The file specified in a %INCLUDE directive contains a record longer than 255 bytes.

User Action: Edit the file to remove any records longer than 255 bytes.

RECTOOLAR, RECORD too large. Limit is 65,535 bytes

Explanation: ERROR—The components of a RECORD definition add up to more than 65,535 bytes, or 32,767 bytes if the RECORD is used as an array component.

User Action: Reduce the size of the RECORD definition.

REMARRREF, entire REMAPPED array <name> cannot be passed BY REF

Explanation: ERROR—The program attempts to pass an array declared in a MAP DYNAMIC statement to an external subprogram by reference.

User Action: Entire remapped arrays must be passed by descriptor. Specify the BY DESC passing mechanism either in the EXTERNAL declaration or the subprogram invocation.

REMNOTALL, REM statement not allowed in programs without line numbers

Explanation: ERROR—A REM statement has been found in a program without line numbers.

User Action: Remove the REM statement.

REPLACE, assuming <operator(s)> replaced by <operator>

Explanation: ERROR—The program contains a syntax error. HP BASIC found incorrect or multiple operators where another single operator makes more sense, for example, 10 A == B. Compilation continues so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

REQNUMEXP, <item> requires a numeric expression

Explanation: ERROR—The program contains a string expression in a context requiring a numeric expression.

User Action: Supply a numeric expression.

REQSTREXP, <item> requires string expression

Explanation: ERROR—The program contains a numeric expression in a context requiring a string expression, for example, the file specification in an OPEN statement or the default file specification in a DEFAULTNAME clause.

User Action: Supply a string expression.

RESABOCON, RESEQUENCE aborted due to conditional compilation

Explanation: ERROR—A resequenced program contains a %IF-%THEN-%ELSE-%END-%IF directive.

User Action: Remove the %IF-%THEN-%ELSE-%END-%IF directive.

RESABOSYN, RESEQUENCE aborted due to syntax error

Explanation: ERROR—A RESEQUENCE operation was terminated because the program was not syntactically correct.

User Action: Correct the syntax error and retry the RESEQUENCE operation.

RESATTINC, result attributes inconsistent with prior declaration

Explanation: ERROR—An external or DEF function declaration specifies a data type for the function's result, which is different from the data type the DEF or FUNCTION statement specifies.

User Action: Change the specified data type in either the declaration or the DEF or FUNCTION statement so that the data types agree.

RESINCLIN, RESEQUENCE cannot be used if INCLUDE files reference line numbers

Explanation: ERROR—The current program references an INCLUDE file that contains line number references, for example, GOTO.

User Action: Remove the %INCLUDE directive. HP BASIC cannot resequence lines in an INCLUDE file.

RESLINGTR, RESEQUENCE cannot generate line numbers greater than 32,767

Explanation: ERROR—The RESEQUENCE command specified an interval or starting point that would have created a line number greater than 32,767.

User Action: Reduce the interval or the starting point.

RESNOTWHE, RESUME not allowed in WHEN block or handler

Explanation: ERROR—A RESUME statement has been found in a WHEN block or an associated error handler.

User Action: Remove the RESUME statement from the WHEN block or associated error handler.

RESORDLIN, RESEQUENCE cannot change the order of or delete lines

Explanation: ERROR—The RESEQUENCE command specifies invalid source program changes.

User Action: Supply a valid RESEQUENCE command.

RETCONMUS, RETRY and CONTINUE must appear in error handlers

Explanation: ERROR—A RETRY or CONTINUE statement is not in an error handler associated with a WHEN block protected region.

User Action: Remove the RETRY or CONTINUE statement.

RFAEXPREQ, RFA expression required

Explanation: ERROR—A GET BY RFA statement contains an expression that is not of the RFA data type.

User Action: Supply a valid RFA expression.

RFANOTALL, RFA not allowed in this context

Explanation: ERROR—The program attempts to use an RFA expression in an arithmetic expression or other invalid context.

User Action: Remove the RFA expression. You can use the RFA data type only in file I/O, in an assignment statement, or in a comparison.

ROUSUPDEC, ROUNDing supported only for DECIMAL

Explanation: ERROR—Rounding was specified for a non-DECIMAL data type in: 1) a compiler command, 2) a qualifier to the BASIC DCL command, or 3) an OPTION statement.

User Action: Specify rounding only for the DECIMAL data type.

RPTCOUMUS, repeat count must be positive numeric

Explanation: ERROR—A FILL item specifies a nonnumeric or negative repeat count, for example, FILL(A$) or FILL(-3).

User Action: Supply a valid repeat count.

SCAFACINH, SCALE factor inhibits optimization

Explanation: INFORMATION—This error is reported only when the /SETUP qualifier is in effect. Specifying a scale factor prevents optimization of the compiler-generated code.

User Action: Compile the program without specifying a scale factor.

SCAFILMUS, ANALYSIS file must be random access—no ANA file produced

Explanation: WARNING— The file specification on the /ANALYSIS_DATA qualifier specifies a nonrandom access device; therefore, no analysis data file will be produced. HP BASIC ignores the /ANALYSIS_DATA qualifier and continues compilation.

User Action: Specify a random access device on the file specification on the /ANALYSIS_DATA qualifier.

SCALE0, scale factor used is 0 for single precision

Explanation: WARNING—An attempt was made to set the SCALE factor while in single precision.

User Action: Set the precision to /REAL_SIZE = DOUBLE. You cannot use scaling when in single precision.

SCANOTANS, /ANALYSIS_DATA qualifier not allowed with /ANSI

Explanation: ERROR—The /ANALYSIS_DATA qualifier conflicts with the /ANSI_STANDARD qualifier.

User Action: Specify either the /ANALYSIS_DATA qualifier or the /ANSI_STANDARD qualifier, but not both.

SCAOUTRAN, SCALE is out of range. Valid is 0 to 6.

Explanation: ERROR—The OPTION statement specifies a scale factor that is not from 0 to 6, inclusive.

User Action: Supply a valid scale factor.

SEQERR, attempt to sequence over existing statement

Explanation: ERROR—A SEQUENCE command specifies a starting line number that already exists in the HP BASIC source program in memory.

User Action: Specify a starting line number higher than any existing line or delete the old statement before using the SEQUENCE command.

SEVERRSCA, please submit an SPR —- internal error in SCA support

Explanation: FATAL—A severe error has been detected in the SCA support in the HP BASIC compiler. If you recompile your program without the /ANALYSIS_DATA qualifier, this error should no longer occur.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces the error.

SEVINTERR, severe internal error has been detected. Submit an SPR.

Explanation: FATAL—An error has been detected in the HP BASIC compiler.

User Action: Please submit a software problem report (SPR) with the source code of a small program that produces this error.

SHRNOTAVL, Unable to access the shareable image <name>

Explanation: ERROR—The shareable image is not available on your system.

User Action: Install the correct version of the required shareable image.

SPANOSPA, SPAN is inconsistent with NOSPAN

Explanation: WARNING—An OPEN statement specifies both SPAN and NOSPAN.

User Action: Remove one of the clauses.

SPELL, assuming <item> intended to be the keyword: <keyword>

Explanation: ERROR—The program contains a syntax error. HP BASIC assumes that a keyword has been misspelled, and compilation continues so that other errors can be detected. The actual program line remains unchanged and no object file is produced.

User Action: Examine the line carefully to discover the error. Change the program line to correct the syntax error.

SPENUMEXC, specified numeric exceeds valid character code

Explanation: ERROR—A quoted literal of type character (C) contains a value outside the valid range, for example, ′300′C.

User Action: Use a valid ASCII value.

STACKOVF, stack frame overflow for variables

Explanation: ERROR—The program requires too much space for dynamic variables.

User Action: Reduce the number of dynamic variables or place some of the variables in a MAP or COMMON.

STANOTALL, statement not allowed within a PICTURE definition

Explanation: ERROR – The statement you specified is not allowed in a PICTURE definition.

User Action: Remove the statement from the PICTURE definition.

STARISNEE, star (*) is needed in DEF, not "/"

Explanation: ERROR—The program contains a statement that starts with DEF/.

User Action: Change the DEF/ to DEF*.

STRARRNOT, string array not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains a string array.

User Action: Remove the string array.

STRCONEXP, string constant expression is too long

Explanation: ERROR—The program contains a DECLARE STRING CONSTANT statement where the value assigned to the constant exceeds the maximum number of characters allowed for string constant expressions. The maximum length of a string constant expression at compile time is 498 characters.

User Action: Change the string constant to a string variable and assign the string expression to the variable at run time.

STRCONREQ, string constant required

Explanation: ERROR—The program contains a numeric expression in a context that requires a string expression, for example:

```
DECLARE STRING CONSTANT ABC = 123
```

User Action: Supply a string literal or a named string constant.

STRDEFNOT, string DEF not ANSI

Explanation: INFORMATION—A program compiled with the /ANSI_ STANDARD qualifier contains a string DEF.

User Action: Remove the string DEF.

STRLENANY, string length not allowed on ANY

Explanation: ERROR—An ANY parameter specifies a string length in an EXTERNAL statement. This is not allowed because ANY implies that you can use any data type, not specifically a string data type.

User Action: Remove the string length specification from the ANY clause.

STRIS_NEE, string expression is required

Explanation: ERROR—The program contains a numeric expression where a string expression is needed, for example, NAME 1% AS "ABC.DAT".

User Action: Supply a string expression.

STRLENDYN, string length not allowed on MAP DYNAMIC variable

Explanation: ERROR—A string variable in a MAP DYNAMIC statement specifies a string length.

User Action: Remove the string length. All string variables named in a MAP DYNAMIC statement have a length of zero until a REMAP statement executes.

STRLENINC, virtual array string <name> length increased from <n> to <m>

Explanation: WARNING—In a string virtual array DIM statement, the specified string length is not a power of two.

User Action: None. HP BASIC increases the string length to the next higher power of two.

STRLENMUS, string length specification for <name> must be numeric

Explanation: ERROR—The length specification for a fixed-length string is nonnumeric, for example, COMMON A$ = "ABC".

User Action: Supply a numeric length specification.

STRLENNOT, string length not allowed on numeric variable <name>

Explanation: ERROR—The declaration for a numeric variable contains a string length specification.

User Action: Remove the string length specification.

STRLENTRU, virtual array string <name> length truncated from <n> to <m>

Explanation: WARNING—A string virtual array specifies a string length greater than 512. HP BASIC truncates the length specification to 512.

User Action: None. The maximum string length for virtual arrays is 512.

STRLITREQ, string literal required for compiler directive

Explanation: ERROR—A quoted string is missing in a compiler directive that requires one, for example, %IDENT.

User Action: Supply a string literal for the compiler directive.

STROUTRAN, string is too large

Explanation: ERROR—A string exceeds the maximum allowable length. The maximum length is 65,535 characters.

User Action: Reduce the length of the string.

STRRECFIE, string record element may not be FIELDed

Explanation: ERROR—A FIELD statement contains a string record element as the fielded variable.

User Action: Replace the string record element with a dynamic string. Fielded variables must be dynamic.

STRRECFOR, stream format must have sequential organization

Explanation: ERROR—A file was opened using STREAM as a record format, but the specified organization was not SEQUENTIAL.

User Action: Change the OPEN statement so that it specifies ORGANIZATION SEQUENTIAL.

STRVAREXP, string variable expected

Explanation: ERROR—A CHANGE statement specifies a numeric variable.

User Action: Supply a string variable; the CHANGE statement changes a string variable to a numeric array and vice versa.

STRVARREQ, string variable required

Explanation: ERROR—A statement references a numeric variable instead of a string variable, for example, LINPUT A%.

User Action: Supply a string variable instead of a numeric variable.

SUBMAYNOT, subscript may not be specified for entire array

Explanation: ERROR—A CALL statement or external function reference passes an entire array as a parameter and contains a subscript expression, for example, A(,,3).

User Action: Remove the subscript expression. You cannot specify any subscripts when passing an entire array as a parameter.

SUBOUTRAN, subscript out of range for <array-name>

Explanation: ERROR—The program references an array element with constant subscript(s) outside the bounds of the array.

User Action: Check program logic to make sure all subscripts are within the bounds of the array.

SUBRECCOM, subscripting error in RECORD component

Explanation: ERROR—The program contains a RECORD component reference with invalid subscripts, for example, A::B(1,2)::C where B has only one subscript, or A::B where A requires a subscript.

User Action: Change the erroneous reference. You must specify as many subscripts as were defined in the RECORD.

SUBWHINOT, exit from SUB seen while not in SUB

Explanation: ERROR—A program contains an EXIT SUB or SUBEXIT statement with no preceding SUB statement.

User Action: If the program is a subprogram, supply a SUB statement; otherwise, remove the EXIT SUB or SUBEXIT statement.

SUFFILNOT, suffix not allowed on FILL after datatype keyword

Explanation: ERROR—A FILL item defined with an explicit data type ends in a percent or dollar sign.

User Action: Remove the FILL item's percent or dollar sign.

SUFINTONLY, % only allowed with BYTE, WORD, LONG, QUAD, or INTEGER keywords

Explanation: ERROR—The % suffix is only allowed on integer data types.

User Action: Remove the % suffix from the variable name or change the data-type keyword.

SUFNOTALL, suffix not allowed on variable <name>

Explanation: ERROR—A name, which cannot end in a percent sign or dollar sign, such as a label name, ends with either a percent sign or dollar sign.

User Action: Remove the variable's percent or dollar sign.

SUFNOTHAN, suffix not allowed on HANDLER <name>

Explanation: ERROR—A HANDLER name ends in a percent or dollar sign.

User Action: Remove the percent or dollar sign from the HANDLER name.

SUFNOTREC, suffix not allowed for record type

Explanation: ERROR—A record definition specifies a user-defined record type that ends in a percent or dollar sign.

User Action: Remove the record type's percent or dollar sign.

SUFSTRONLY, $ is only allowed with STRING keyword

Explanation: ERROR—The $ suffix is only allowed on string data types.

User Action: Remove the $ suffix from the variable name or change the data-type keyword.

SYNNOTANS, syntax check mode not allowed when ANSI

Explanation: ERROR—A SET /SYNTAX_CHECK command was entered when the /ANSI_STANDARD qualifier was in effect.

User Action: None; syntax checking is not supported in ANSI mode.

SYSERROR, system service error

Explanation: ERROR—An error was detected while executing a system service.

User Action: Take corrective action based on the associated message.

TEXFOLEND, text following end of program unit must be on new <type of line> line

Explanation: ERROR—The compiler detected text following an END, END SUB, or END FUNCTION statement.

User Action: Remove the text. In a multimodule source file with line numbers, any text following an END, END SUB, or END FUNCTION statement must begin on a numbered line. In a multimodule source file without line numbers, any text following an END, END SUB, or END FUNCTION statement must begin on a new physical line.

TEXLINMSG, text line exceeded 255 characters

Explanation: INFORMATION—An input line contains more than 255 characters. HP BASIC saves the first 255 input characters into the line buffer and ignores the rest of the input.

User Action: Supply no more than 255 characters per input line to avoid truncation of input.

TEXPATMUS, text path must be “RIGHT”, “LEFT”, “UP” or “DOWN”

Explanation: ERROR—You specified an invalid value for the path specification of the SET TEXT PATH statement.

User Action: Specify one of the values listed in the message.

TEXPREMUS, text precision must be “STROKE”,“CHAR” or “STRING”

Explanation: ERROR—You specified an invalid value for the text precision of the SET TEXT FONT statement.

User Action: Specify one of the values listed in the message.

THEMUSFOL, THEN directive must follow a lexical expression

Explanation: ERROR—A %IF directive contains a lexical expression that is not immediately followed by a %THEN clause.

User Action: Supply a %THEN clause. %THEN, %ELSE, and %END %IF are required in a %IF directive.

TOMCHINFO, extra information about command line has been ignored

Explanation: INFORMATION—You supplied an argument to a CONTINUE, EXIT, IDENTIFY, or SCRATCH command. These commands do not accept arguments. HP BASIC ignores the extra data and executes the command.

User Action: Remove the argument from the command.

TOOFEWARG, too few arguments

Explanation: ERROR—The invocation of an HP BASIC built-in function contains too few arguments.

User Action: Supply the correct number of arguments to the function.

TOOMANARG, too many arguments

Explanation: ERROR—The invocation of an HP BASIC built-in function contains too many arguments.

User Action: Supply the correct number of arguments to the function.

TOOMANIND, too many array indices active

Explanation: ERROR—A subscript expression contains more than 100 array indices between the open parenthesis and the close parenthesis.

User Action: Reduce the number of active array indices.

TOOMANKEY, too many keys—limit is 255

Explanation: ERROR—An OPEN statement specifies more than 255 index keys.

User Action: Reduce the number of index keys. The maximum is 255.

TOOMANPAR, too many function parameters active

Explanation: ERROR—An external function invocation contains too many expressions in the actual parameter list.

User Action: Reduce the number of expressions in the actual parameter by assigning the expressions to temporary variables.

TRAFUNONL, Transformation functions only permitted with multiplication

Explanation: ERROR—A graphics transformation function is used in a MAT statement other than matrix multiplication.

User Action: Remove the transformation function from the MAT statement.

TRAOUTRAN, transformation number must be in the range 1 - 255

Explanation: ERROR—You specified a transformation number that is less than 1 or greater than 255.

User Action: Change the transformation number to be within the range 1 to 255.

TYPDEFSTR, TYPE default of STRING is not allowed

Explanation: ERROR—STRING was specified as the default data type in: 1) a compiler command, 2) a qualifier to the DCL BASIC command, or 3) an OPTION statement.

User Action: Specify a numeric data type as the default.

UNCALLED, Routine <routine> can never be called

Explanation: INFORMATION—The compiler has detected a routine that is never called.

User Action: Review the program to determine if the routine is needed. If it is not, you may want to remove it.

UNDEFINED, unresolved/undefined symbols

Explanation: ERROR—A program executed in the VAX BASIC Environment calls or invokes a subprogram or routine that has not been loaded.

User Action: Load the subprogram or routine before running the program in the VAX BASIC Environment.

UNDLINNUM, undefined line number

Explanation: ERROR—A statement tries to transfer control to a nonexistent line. Or, in a numberless program, a line number is referenced.

User Action: Replace the nonexistent line number with the correct destination line number or label.

UNELEXDIR, unexpected lexical directive encountered

Explanation: ERROR—The specified lexical directive is not legal in this statement.

User Action: Use a supported lexical directive.

UNEXPEOF, unexpected end of file

Explanation: ERROR—An end-of-file was specified immediately after an ampersand continuation character.

User Action: Remove the ampersand continuation character or continue the line.

UNINIT, variable <variable> is fetched, not initialized

Explanation: INFORMATION—The compiler has detected a variable that is used but not initialized.

User Action: Review the program to determine if the variable should be initialized before use. If necessary, you may want to add code to initialize the variable.

UNKCOMINP, unknown command input

Explanation: ERROR—An attempt was made to enter an invalid or unknown command.

User Action: Enter the HP BASIC command correctly.

UNLINCREA, UNLOCK EXPLICIT clause inconsistent with ACCESS READ

Explanation: ERROR—An OPEN statement contains both an ACCESS READ and an UNLOCK EXPLICIT clause. This is inconsistent because ACCESS READ specifies no record locking while UNLOCK EXPLICIT specifies that all accessed records remain locked until explicitly unlocked.

User Action: Either remove the UNLOCK EXPLICIT clause or change the ACCESS clause.

UNREACH, code can never be executed at label <label>

Explanation: INFORMATION—The compiler has detected code that will never be executed, for example, a multistatement line whose first statement is a GOTO, EXIT, ITERATE, RESUME, or RETURN. (Alpha BASIC only)

User Action: Review the program to determine if the code should be executed. If you determine the code should be executed, then you should revise the program flow logic accordingly; otherwise, the code is unnecessary, and you may want to remove it. In the case of the GOTO, EXIT, ITERATE, RESUME, or RETURN statements, make sure that these statements are the only statements on the line, or the last statement on a multistatement line.

UNSCDDLEV, unsupported CDD/Repository level <number>. Supported level is <number>.

Explanation: ERROR—The current CDD/Repository version is incompatible with HP BASIC.

User Action: Use a supported version of CDD/Repository.

UNTSTRLIT, unterminated string literal

Explanation: ERROR—The program contains an improperly terminated string literal; for example, "ABC , "ABC′, and ′ABC" are all improperly terminated.

User Action: Use the same type of quotation mark (either single or double) for both beginning and ending string delimiters.

USEONLALO, USE only allowed inside WHEN blocks

Explanation: ERROR—A USE statement is not within a WHEN block.

User Action: Remove the USE statement.

USERABORT, user ABORT directive <text>

Explanation: FATAL—The compilation was terminated as the result of a %ABORT directive. The compiler prints the text following the %ABORT.

User Action: None.

USERPRINT, <text>

Explanation: SUCCESS—The compilation found a %PRINT directive and printed the specified message to the terminal and listing file.

User Action: None.

USEVARNOT, user variable <name> not allowed in declaration

Explanation: ERROR—The parameter list in an external subprogram declaration contains a user variable name.

User Action: Remove the variable from the parameter list. When declaring a routine, the parameter list can contain only data type and parameter-passing mechanism specifications.

VALTOOLAR, value too large for constant

Explanation: WARNING—The value of an EXTERNAL CONSTANT is larger than the specified data type allows.

User Action: Make sure the data type specified in the EXTERNAL CONSTANT statement matches that of the actual constant.

VALUEREQ, PRINT USING requires a value

Explanation: ERROR—A PRINT USING statement must have at least one expression or value.

User Action: Supply an expression or value at the end of the PRINT USING statement.

VARCONREQ, variable or constant required

Explanation: ERROR—The program contains an executable DIM statement that contains an expression in the bounds list.

User Action: Remove the expression from the bounds list. Executable DIM statements can have only constants or variables (simple or subscripted) as bounds.

VARNOALGN, Variable <name> within COMMON or MAP is not naturally aligned.

Explanation: WARNING—Identifies a variable within a COMMON or MAP that was found not to be naturally aligned. This error is only reported when the /WARNING=ALIGNMENT qualifier is in effect.

User Action: Modify COMMON or MAP so that all variables are naturally aligned.

VERJUSMUS, vertical justification must be “TOP”, “CAP”,“HALF”,“BASE”, “BOTTOM” or “NORMAL”

Explanation: ERROR—You specified an invalid value for the vertical component of the SET TEXT JUSTIFY statement.

User Action: Specify one of the values listed in the message.

VIRARROVF, virtual array space exceeded at array <name>

Explanation: ERROR—The storage for virtual arrays on a single channel exceeds 2,147,483,647 bytes.

User Action: If there is only one virtual array on the channel, you must reduce the amount of storage used by the array. However, if there is more than one virtual array on the channel, you can put each array on a separate channel.

VIRNOTALL, virtual array not allowed in graphics statements

Explanation: ERROR—You specified an entire virtual array on a statement that does not allow them.

User Action: Specify a nonvirtual array in place of the virtual array.

VIRRECTOO, virtual RECORD <name> is too large. Limit is 512 bytes

Explanation: ERROR—The elements of a virtual array are of type <name> and the total storage requirement for each element is greater than 512 bytes.

User Action: Reduce the size of the RECORD.

WRITEERR, error writing <file-name>

Explanation: ERROR—An error was detected in attempting to write to a file.

User Action: Supply a valid file specification or take corrective action based on the associated message.

WROTYPLIB, library <lib-name> is not an OBJECT or IMAGE library

Explanation: WARNING—The logical BASIC$LIBn translates to a library that is not an object library or a shareable image library.

User Action: Change the logical BASIC$LIBn to translate to an object library or a shareable image library.

XYPOIREQ, X,Y point required between semicolons

Explanation: ERROR—In a list of points in a statement such as PLOT LINES, you specified two semicolons in a row without an X,Y point specification between them.

User Action: Either supply another point or remove the extra semicolon.

B
Run-Time Messages

HP BASIC returns run-time messages if an error occurs while a program is executing. The error is diagnosed and for programs without line numbers, HP BASIC indicates the program line that generated the error. Warning messages indicate that an error has occurred, but program execution continues.

In some cases, HP BASIC performs the specified operation, but the results are not as expected. Fatal (severe) error messages indicate that the program has aborted. You can recover from most fatal errors by including error-handling routines in your program and by specifying OPTION HANDLE = SEVERE. Certain errors, however, are not recoverable even when error-handlers are used. In the descriptions of these errors, they are designated as not able to be trapped. You do not need error-handling routines to trap errors that generate warning messages.

Section B.1 lists HP BASIC run-time errors, alphabetized by mnemonic code. Section B.2 is a cross-reference numerical listing of run-time errors generated by HP BASIC. Section B.3 lists messages that HP BASIC does not generate, but which can be displayed with the ERT$ function. See the *HP BASIC for OpenVMS Reference Manual* for information about RMSSTATUS and VMSSTATUS.

B.1 HP BASIC Run-Time Errors by Mnemonic

The HP BASIC error message format is as follows:

%BAS-<I>-<mnemonic>, <message> -BAS-I-FROLINMOD, from Line *x* in module *y*

<I>
Is a letter indicating the severity of the error. The severity indicator can be one of the following:

- I—Indicating information
- W—Indicating a warning

- E—Indicating an error
- F—Indicating a severe error

<mnemonic>
Is a 3- to 9-character string that identifies the error.

<X>
Is the line number where the error occurred.

<Y>
Is the name of the module where the error occurred.

Warning error messages indicate that an error has occurred, but program execution continues. In some cases, HP BASIC reprompts for more information or correct data; in other cases, HP BASIC performs the specified operation, but the results are not as expected. Fatal error messages indicate that the program has aborted.

ARGDONMAT, Arguments don't match (ERR=88)

Explanation: The proper array descriptor was not specified for a matrix operation.

User Action: Use HP BASIC to create the array.

ARGTOOLAR, Argument too large in EXP (ERR=49)

Explanation: The program contains:

- An argument to the EXP function larger than 88
- An exponentiation operation that results in a number greater than 1E38

User Action: Change the EXP argument to be in the valid range, or reduce the size of the exponent.

ARRMUSSAM, Arrays must be same dimension (ERR=238)

Explanation: The program attempts to perform matrix addition or subtraction on input arrays with a different dimensions.

User Action: Use arrays that have identical dimensions.

ARRMUSSQU, Arrays must be square (ERR=239)

Explanation: The program attempts matrix inversion (MAT INV) on an array that is not inversible.

User Action: Use only square arrays when performing a matrix inversion.

ARRTOOSMA, Array too small (ERR=197)

Explanation: The array you referenced with a graphics statement is too small. Check the description of the graphics statement to get the minimum size requirement for the array.

User Action: Increase the size of the array.

BADDIRDEV, Bad directory for device (ERR=1)

Explanation: The device directory does not exist or is unreadable.

User Action: Supply a valid directory.

BADRECIDE, Bad record identifier (ERR=143)

Explanation: The program attempted a record access that specified:

- A zero or negative record number on a RELATIVE file
- A null key value on an INDEXED file

User Action: Change the record number or key specification to a valid value.

BADRECVAL, Bad RECORDSIZE value on OPEN (ERR=148)

Explanation: The value in the RECORDSIZE clause in the OPEN statement either 1) is zero or greater than 65,535 or 2) does not match the record size of an existing file.

User Action: Change the value in the RECORDSIZE clause.

CANCHAARR, Cannot change array dimensions (ERR=240)

Explanation: The program attempts to redimension an array to a different number of dimensions.

User Action: Change the arrays dimensions in the DIM or MAT statement.

CANFINFIL, Cannot find file or account (ERR=5)

Explanation: The specified file or directory is not present on the device.

User Action: Supply a valid file specification.

CANINVMAT, Cannot invert matrix (ERR=56)

Explanation: The program attempts to invert a single-dimension matrix.

User Action: Supply a matrix of the proper form for inversion.

CANOPEFIL, Cannot open file (ERR=162)

Explanation: The program attempts to open a file that cannot be opened.

User Action: Use VMSSTATUS to determine the RMS failure that caused the error.

CLIPONOFF, Clipping must be set to ON or OFF (ERR=259)

Explanation: Valid strings for the SET CLIP statement are "ON" and "OFF."

User Action: Change the string to either "ON" or "OFF."

COLNOTCON, Color indices are not contiguous (ERR=261)

Explanation: The color indices on the device you are using are not contiguous.

User Action: Unlike most devices, all color indices between zero and the number returned by the ASK MAX COLOR statement are not available on this device.

COONOTNDC, Coordinates are not within NDC space (ERR=273)

Explanation: The boundaries of NDC space are 0,1,0,1; coordinates must be within this range.

User Action: Supply coordinates with values from 0 to 1. Make sure that the minimum value of x is less than the maximum value of x and that the minimum value of y is less than the maximum value of y.

CORFILSTR, Corrupted file structure (ERR=29)

Explanation: RMS has detected an invalid file structure on disk.

User Action: See your system manager.

DATFORERR, Data format error (ERR=50)

Explanation: The program specifies a data type in a statement that does not agree with the value supplied or invalid data was used in string arithmetic.

User Action: Change the statement or supply data of the correct type.

DATOVERF, data overflow (ERR=289)

Explanation: The keystroke retrieved by the INKEY$ function caused the type-ahead buffer to overflow or the terminal attempted to send a valid ANSI escape sequence that did not correspond to a keystroke.

User Action: Specify the DCL command SET TERMINAL/HOSTSYNC before using the INKEY$ function. This command will prevent the type-ahead buffer from overflowing.

DATTYPERR, Data type error (ERR=101)

Explanation: The program attempts to access a parameter passed BY DESC (by descriptor), and the descriptor contains an incorrect data type. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Check the program code that created the passed parameter and make sure it creates a parameter of correct data type.

DEADLOCK, Detected deadlock while waiting for GET or FIND (ERR=193)

Explanation: The record your program is trying to access is currently locked on another channel or by another process. Simultaneously, your program has locked a record that the other user cannot access. The deadlock cannot be resolved.

User Action: Possible solutions include:

- Use the FREE statement to unlock all locked records
- Use GET...REGARDLESS if read access is sufficient

DECERR, DECIMAL error or overflow (ERR=181)

Explanation: The result of a DECIMAL expression is greater than or requires more precision than can be contained in the variable.

User Action: Reduce the magnitude of the expression or increase the allowed digits in the DECIMAL variable.

User Action: Check program logic or trap the error in an error handler.

DEVHUNWRI, Device hung or write locked (ERR=14)

Explanation: The program attempted an operation to a hardware device that is not functioning properly or is protected against writing.

User Action: Check the device on which the operation is performed.

DEVINMET, Device is an input metafile (ERR=270)

Explanation: The operation cannot be performed on an input metafile (device type 3).

User Action: Specify the device ID for a device other than an input metafile.

DEVNOTOPE, Device is not open (ERR=268)

Explanation: The device has not been identified in an OPEN...FOR GRAPHICS statement.

User Action: Specify the device ID number in an OPEN...FOR GRAPHICS statement.

DEVOPEINC, Device and operation are incompatible (ERR=272)

Explanation: The operation you requested cannot be performed on the specified device. For example, output cannot be dislayed on a device that is for input only.

User Action: Specify the device ID for a device with the appropriate compatibility. Device types are listed in *Programming with VAX BASIC Graphics*.

DEVOUTMET, Device is an output metafile (ERR=269)

Explanation: The specified device is an output metafile (device type 2).

User Action: Specify the device ID for a device other than an output metafile.

DEVTYPNOT, Device type is not supported (ERR=267)

Explanation: The specified device type is not supported by Compaq GKS for OpenVMS.

User Action: Specify an alternative device type. Standard supported device types are listed in *Programming with VAX BASIC Graphics* and in the Compaq GKS for OpenVMS documentation. Verify with your system manager that support for the specified device has been installed. Also, verify that the Compaq GKS for OpenVMS startup command procedure has properly executed.

DIFUSELON, Differing use of LONG/WORD or SINGLE/DOUBLE qualifiers (ERR=229)

Explanation: The main and subprograms were compiled with different LONG/WORD modes. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Recompile one of the programs with the same qualifier as the other.

DIMOUTRAN, Dimension number out of range (ERR=195)

Explanation: The upper or lower bound of the specified dimension cannot be returned because the array has fewer dimensions than the one requested.

User Action: Change the dimensions specified with the LBOUND or UBOUND function.

DIRERR, Directive error (ERR=253)

Explanation: A system service call resulted in an error.

User Action: See the *VMS I/O User's Reference Volume* or the *OpenVMS Record Management Services Reference Manual*.

DIVBY_ZER, Division by 0 (ERR=61)

Explanation: The program attempts to divide a value by zero.

User Action: Check program logic and change the attempted division or trap the error in an error handler.

DUPKEYDET, Duplicate key detected (ERR=134)

Explanation: In a PUT operation to an indexed file, a duplicate key was specified, and DUPLICATES was not specified when the file was created.

User Action: Change the duplicate key, or recreate the file specifying DUPLICATES for that key.

ECHTYPNOT, Prompt/echo type not supported (ERR=256)

Explanation: The specified prompt or echo type is invalid. HP BASIC supports only the default prompt and echo types.

User Action: Do not change the prompt or echo type. If you do so, you should continue to use direct calls to Compaq GKS routines rather than use HP BASIC input statements.

ENDFILDEV, End of file on device (ERR=11)

Explanation: The program attempted to read data beyond the end of the file.

User Action: None. The program can trap this error in an error handler.

ENTPOINOT, Entered points not within a transformation (ERR=285)

Explanation: Input points are not within the viewport of a defined transformation.

User Action: Issue a warning to the user to input points within the defined area. Alternatively, you can change at least one transformation to include the viewport area not defined. At the start of program execution, transformation 1 includes all of NDC space. Optionally, you can define one transformation to cover the default viewport.

ERRFILCOR, Error on OPEN - file corrupted (ERR=178)

Explanation: The program attempted to open an invalid structure on disk.

User Action: See your system manager.

ERRTRANEE, ERROR trap needs RESUME (ERR=246)

Explanation: An error handler attempts to execute an END, END SUB, END FUNCTION, SUBEND, FUNCTIONEND, or FNEND statement without first executing a RESUME statement. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Change the program logic so that the error handler executes a RESUME statement before executing an END, END SUB, END DEF, SUBEND, FUNCTIONEND, or FNEND statement.

FATSYSIO_, Fatal system I/O failure (ERR=12)

Explanation: An I/O error has occurred in: 1) the system or 2) Record Management Services. The last operation will not be completed.

User Action: See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for RMS errors or retry the operation. Use VMSSTATUS to return the VMS condition code that caused the error.

FIEOVEBUF, FIELD overflows buffer (ERR=63)

Explanation: A FIELD statement attempts to access more data than exists in the specified buffer.

User Action: Change the FIELD statement to match the buffer's size, or increase the buffer's size.

FILACPFAI, FILE ACP failure (ERR=252)

Explanation: The operating system's file handler reported an error to RMS.

User Action: See the *VMS I/O User's Reference Volume* or the *OpenVMS Record Management Services Reference Manual*.

FILATTNOT, File attributes not matched (ERR=160)

Explanation: The following attributes in the OPEN statement do not match the corresponding attributes of the target file:

- ORGANIZATION
- BUCKETSIZE
- BLOCKSIZE
- Key number, size, position, or attributes (CHANGES and DUPLICATES)
- Record format

User Action: Change the OPEN statement attributes to match those of the file or remove the clause.

FILEXPDAT, File expiration date not yet reached (ERR=174)

Explanation: The program attempted to delete a file before the file's expiration date was reached.

User Action: Change the file's expiration date.

FILIS_LOC, File is locked (ERR=138)

Explanation: The program does not allow shared access, and attempts to access a file that has been locked by another user or by the system.

User Action: Change the OPEN statement to allow shared access or wait until the file is released by other users.

FLOPOIERR, Floating point error or overflow (ERR=48)

Explanation: A program operation resulted in a floating-point number with absolute value outside the allowable range for that data type.

User Action: Check program logic or trap the error in an error handler.

FNEWITFUN, FNEND without function call (ERR=73)

Explanation: The program executes an END DEF or FNEND statement before executing a function call. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Check program logic to make sure that END DEF or FNEND statements are executed only in multiline DEFs or remove the END DEF or FNEND statement.

GKSNOTINS, DEC GKS FOR VMS is not installed (ERR=226)

Explanation: Graphics statements are not operational when Compaq GKS is not installed.

User Action: See your system manager.

ILLALLCLA, Illegal ALLOW clause (ERR=168)

Explanation: The value specified for the ALLOW clause (sharing) on the OPEN statement is illegal for the type of file organization.

User Action: Change the ALLOW clause argument.

ILLARGLOG, Illegal argument in LOG (ERR=53)

Explanation: The program contains a negative or zero argument to the LOG or LOG10 function.

User Action: Supply an argument in the valid range.

ILLARESTY, Illegal area style (ERR=262)

Explanation: Area style must be one of the following:

- SOLID (the default)
- HOLLOW
- HATCH
- PATTERN

User Action: Specify a valid area style for the device.

ILLBYTCOU, Illegal byte count for I/O (ERR=31)

Explanation: A PRINT or INPUT list invoked a function that closed an I/O channel.

User Action: Change the function so that it does not close the I/O channel.

ILLCNTCLA, Illegal count clause (ERR=290)

Explanation: In a graphics statement, you specified a COUNT clause with a numeric value that exceeds the size of the array.

User Action: Specify a numeric value that is less than or equal to the size of the array.

ILLCOLIND, Illegal color index (ERR=280)

Explanation: The index you specified is not supported by the device.

User Action: Specify a valid color index. The valid range of indices for the device is from 0 to the value retrieved by the ASK MAX COLOR statement.

ILLCOLMIX, Illegal color mix (ERR=291)

Explanation: The color mix value specified on the SET COLOR MIX statement is outside the range of 0 to 1.

User Action: Specify a value from 0 to 1.

ILLDEVID, Illegal device identification number (ERR=266)

Explanation: The device identification number is beyond the valid range of 0 through 255.

User Action: Specify a device identification number between 0 and 255.

ILLDEVNAM, Illegal device name in OPEN (ERR=292)

Explanation: An explicit or implicit OPEN...FOR GRAPHICS statement contains an illegal device name for the device type being used. Possible causes include:

- Specifying a device that does not exist on the system
- Specifying a logical name that is not defined
- Specifying a file name that does not exist when the device type is for an input metafile
- Specifying a file name for a device type that requires an OpenVMS physical device name

User Action: Specify an appropriate device name.

ILLECHARE, Illegal echo area (ERR=283)

Explanation: The specified echo area boundaries are invalid.

User Action: Specify echo area boundaries within the device viewport.

ILLEXIDEF, Illegal exit from DEF* (ERR=245)

Explanation: A multiline DEF* contains a branch to an END, END SUB, END DEF, SUBEND, or FUNCTIONEND statement. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Change the program logic so that the program executes the multiline function's END DEF or FNEND statement before executing the END, END SUB, END DEF, SUBEND, or FUNCTIONEND statement.

ILLFIEVAR, Illegal FIELD variable (ERR=122)

Explanation: A FIELDed variable is referenced after a non-BASIC subprogram closed the file associated with that variable. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Check program logic; do not reference the variable after the file has been closed.

ILLFILNAM, Illegal file name (ERR=2)

Explanation: A file name is: 1) too long, 2) incorrectly formatted, or 3) contains embedded blanks or invalid characters.

User Action: Supply a valid file specification.

ILLILLACC, Illegal or illogical access (ERR=136)

Explanation: The requested access is impossible because:

- The attempted record operation and the ACCESS clause in the OPEN statement are incompatible.
- The ACCESS clause is inconsistent with the file organization.
- ACCESS READ or APPEND was specified when the file was created.

User Action: Change the ACCESS clause.

ILLINIVAL, Illegal initial value (ERR=284)

Explanation: The current initial value specified on the SET INITIAL VALUE or LOCATE VALUE statement is beyond the range of possible values.

User Action: Specify an initial value within the default range (0 through 1) or within the alternative range you optionally specified, or change the range limits.

ILLIO_CHA, Illegal I/O channel (ERR=46)

Explanation: The program specified an I/O channel outside the legal range.

User Action: Specify I/O channels in the range 1 to 99, inclusive or one returned from LIB$GET_LUN.

ILLKEYATT, Illegal key attributes (ERR=137)

Explanation: The program specified CHANGES for the primary key.

User Action: Remove the CHANGES specification from the primary key. You can specify CHANGES only for alternate keys.

ILLLINSIZ, Illegal line size (ERR=275)

Explanation: The specified line size is less than or equal to zero.

User Action: Specify a line size value greater than zero.

ILLLINSTY, Illegal line style number (ERR=274)

Explanation: The specified line style number is less than or equal to zero.

User Action: Specify a valid line style value greater than zero.

ILLNETOPE, Illegal network operation (ERR=190)

Explanation: The program tries to mix GET and PUT operations, or PRINT and INPUT operations, on a remote terminal-format file.

User Action: Change the file organization when opening the file to be sequential variable.

ILLNUM, Illegal number (ERR=52)

Explanation: A value supplied to a numeric variable is incorrect, for example, “ABC” and “1..2” are illegal numbers.

User Action: Supply numeric values of the correct form.

ILLOPE, Illegal operation (ERR=141)

Explanation: The program attempts to:

- DELETE a record in a sequential file.
- UPDATE a record on a magtape file.
- Rewind a process-permanent file.
- DELETE a record in a read-only file.
- Assign a value to a virtual array element in a read-only file.
- Perform a MARGIN operation on VIRTUAL file.

- Transpose a matrix, or perform a matrix multiplication, with the same array as source and destination.
- Perform an invalid operation on a VIRTUAL file, for example, using GET and PUT on a VIRTUAL file, then attempting to reference a virtual array dimensioned on that file.

User Action: Change the illegal operation.

ILLPICOPE, Illegal picture operation (ERR=258)

Explanation: The program attempts to change a transformation within a picture definition. The following statements are invalid within pictures and within routines that are called by pictures:

- SET WINDOW
- SET VIEWPORT
- SET DEVICE WINDOW
- SET DEVICE VIEWPORT
- SET TRANSFORMATION
- SET INPUT PRIORITY
- SET CLIP

User Action: Remove any invalid statements from the picture definition. Set the boundaries for windows and viewports before a picture is invoked.

ILLPOISTY, Illegal point style number (ERR=276)

Explanation: The specified point style is less than or equal to zero.

User Action: Specify a valid point style greater than or equal to zero.

ILLRECACC, Illogical record accessing (ERR=152)

Explanation: The program attempts to perform an operation that is invalid for the specified file type, for example, a random access on a sequential file.

User Action: Supply a valid operation for that file type or change the file type.

ILLRECFIL, Illegal record on file (ERR=142)

Explanation: A record contains an invalid byte count field.

User Action: Use the DCL command DUMP to check the file for possible bad data.

ILLRECLOC, illegal record locking (ERR=187)

Explanation: The program contains an ALLOW clause on a GET statement and the file was not opened with the UNLOCK EXPLICIT clause. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Either remove the ALLOW clause from the GET statement or use the UNLOCK EXPLICIT clause in the OPEN statement.

ILLRESSUB, Illegal RESUME to subroutine (ERR=247)

Explanation: While in an error handler activated by an ON ERROR GO BACK, the error handler attempts to RESUME without a line number. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: None; you cannot specify the RESUME statement without a line number in any program module except in the program module containing the error handler.

ILLSTYIND, Illegal area style index (ERR=279)

Explanation: The specified area style index is less than or equal to zero.

User Action: Specify a valid area style index greater than zero.

ILLSWIUSA, Illegal switch usage (ERR=67)

Explanation: The program attempts an illegal SYS call.

User Action: See the appropriate RSTS/E SYS call documentation.

ILLSYSUSA, Illegal SYS usage( ) (ERR=18)

Explanation: The program attempted an illegal SYS call.

User Action: See the appropriate RSTS/E SYS call documentation.

ILLTEXHEI, Illegal text height (ERR=278)

Explanation: The text height is less than or equal to zero.

User Action: Specify a text height greater than zero.

ILLTEXJUS, Illegal text justification (ERR=263)

Explanation: The specified text justification factor is invalid.

User Action: See *Programming with VAX BASIC Graphics* for valid justification values. Specify valid values.

ILLTEXPAT, Illegal text path (ERR=265)

Explanation: The specified text path is invalid.

User Action: Specify a valid text path. Valid text path values are as follows:

- RIGHT (the default)
- LEFT
- UP
- DOWN

ILLTEXPRE, Illegal text precision (ERR=264)

Explanation: The specified precision string is invalid.

User Action: Valid precision values are: "STROKE" for software fonts, "STRING" and "CHAR" for hardware fonts. Specify a valid string for the precision value.

ILLTEXRAT, Illegal text width-to-height ratio (ERR=276)

Explanation: The specified width-to-height ratio is less than or equal to zero.

User Action: Specify a width-to-height ratio greater than zero.

ILLTFFOPE, Illegal terminal-format file operation (ERR=191)

Explanation: The program specifies a GETRFA function on a terminal-format file.

User Action: Change the file organization when opening the file to be sequential variable.

ILLTRANUM, Illegal transformation number (ERR=257)

Explanation: The specified tranformation number is less than 1 or greater than 255.

User Action: Specify a transformation number from 1 to 255.

ILLUSADEV, Illegal usage for device (ERR=133)

Explanation: The requested operation cannot be performed because:

- The device specification contains illegal syntax.
- The specified device does not exist on your system.

- The specified device is inappropriate for the requested operation (for example, an indexed file access on magnetic tape).

User Action: Supply the correct device type.

ILLWAIVAL, Illegal wait value (ERR=192)

Explanation: The specified integer expression on the WAIT clause is less than zero or greater than 255.

User Action: Specify an integer expression whose value is 0 through 255.

IMASQUROO, Imaginary square roots (ERR=54)

Explanation: An argument to the SQR function is negative.

User Action: Supply arguments to the SQR function that are greater than or equal to zero.

IMPERRHAN, improper error handling (ERR=186)

Explanation: After an error has occurred, a program's error handler calls another program unit, and the called program unit executes an ON ERROR GO BACK statement before clearing the error with a RESUME statement. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Change the program logic so that the called program clears the error condition before executing the ON ERROR GO BACK statement.

INDNOTFUL, Index not fully optimized (ERR=170)

Explanation: A record was successfully written to an INDEXED file; however, the alternate key path was not optimized. This slows record access.

User Action: Delete the record and rewrite it.

INTERR, Integer error (ERR=51)

Explanation: The program contains an integer whose absolute value is greater than 127 in BYTE mode, 32,767 in WORD mode, 2,147,483,647 in LONG mode, or 9,223,372,036,854,775,807 in QUAD mode.

User Action: Use an integer in the valid range for specified data type.

INVCHASTR, Invalid character in string (ERR=287)

Explanation: The program attempts to output a string that contains an invalid character.

User Action: Remove the invalid character from the string.

INVFILOPT, Invalid file options (ERR=139)

Explanation: The program has specified invalid file options in the OPEN statement.

User Action: Change the invalid file options.

INVKEYREF, Invalid key of reference (ERR=144)

Explanation: The program attempts to perform a GET, FIND, or RESTORE on an INDEXED file using an invalid KEY, for example, an alternate KEY that does not exist for the file that was opened.

User Action: Use a valid KEY in the GET, FIND, or RESTORE statement.

INVRFAFIE, Invalid RFA field (ERR=173)

Explanation: During a FIND or GET by RFA, an invalid record's file address was specified.

User Action: Supply a correct RFA field.

IO_CHAALR, I/O channel already open (ERR=7)

Explanation: The program attempted to open a channel in a def or function while I/O was pending on that channel.

User Action: Change the function so that it does not open the channel.

IO_CHANOT, I/O channel not open (ERR=9)

Explanation: The program attempted to perform an I/O operation before opening the channel.

User Action: Open the channel before attempting an I/O operation to it.

KEYFIEBEY, Key field beyond end of record (ERR=151)

Explanation: The position given for the key field exceeds the maximum size of the record.

User Action: Specify a key field within the record.

KEYLARTHA, Key larger than record (ERR=159)

Explanation: The key specification exceeds the maximum record size.

User Action: Reduce the size of the key specification.

KEYNOTCHA, Key not changeable (ERR=130)

Explanation: An UPDATE statement attempted to change a key field that did not have CHANGES specified in the OPEN statement.

User Action: Remove the changed key field in the UPDATE statement or specify CHANGES for that key field in the OPEN statement. Note that the primary key cannot be changed and that you cannot specify CHANGES when you open an existing file if the OPEN statement that created the file did not specify CHANGES.

KEYSIZTOO, Key size too large (ERR=145)

Explanation: The key length on a GET or FIND is either zero or larger than the key length defined for the target record.

User Action: Change the key specification in the GET or FIND statement.

KEYWAIEXH, Keyboard wait exhausted (ERR=15)

Explanation: No input was received during the execution of an INPUT, LINPUT, or INPUT LINE statement that was preceded by a WAIT statement or INKEY$ timeout value.

User Action: None; you must supply input within the specified time.

LINTOOLON, Line too long (ERR=47)

Explanation: The program attempted to input more data than the input buffer can hold. The default input buffer size for terminal input is 132.

User Action: Either decrease the amount of data entered at one time, or increase the size of the input buffer. You can explicitly OPEN the input device and specify the input buffer size with the RECORDSIZE or MAP clause.

MATDIMERR, Matrix dimension error (ERR=124)

Explanation: The program attempts to:

- Assign more than two dimensions to an array.
- Reference an array with fewer or more subscripts than there are dimensions in the array.
- Redimension an array that cannot be redimensioned.
- Perform a matrix operation with an array that has a lower bound, other than zero, in any of its dimensions.

This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Change the number of array subscripts. Reference the array using the correct number of dimensions, change the array so that it can be redimensioned, or change the array so that its lower bounds are zero in all of its dimensions.

MAXMEMEXC, Maximum memory exceeded (ERR=126)

Explanation: The program has insufficient string and I/O buffer space because: 1) its allowable memory size has been exceeded, or 2) the system's maximum memory capacity has been reached. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Reduce the amount of string or I/O buffer space, or split the program into two or more programs.

MEMMANVIO, Memory management violation (ERR=35)

Explanation: The program attempted to read or write to a memory location to which it was not allowed access. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: If the program was compiled with /NOCHECK, it may be exceeding an array bound; recompile with /CHECK. Otherwise, check program logic.

MISSPEFEA, Missing special feature (ERR=66)

Explanation: The program attempts to use an unavailable SYS call.

User Action: See the appropriate RSTS/E SYS call documentation.

MOVOVEBUF, Move overflows buffer (ERR=161)

Explanation: In a MOVE statement, the combined length of elements in the I/O list exceeds the size of the record just read or the size of the buffer.

User Action: Reduce the size of the I/O list or increase the file's RECORDSIZE.

NEGFILSTR, Negative fill or string length (ERR=166)

Explanation: A MOVE statement I/O list contains a FILL item or string length with a negative value.

User Action: Change the FILL item or string length value to be greater than or equal to zero.

NEGZERTAB, negative or zero TAB (ERR=176)

Explanation: The program attempted a zero or negative TAB. This error is signaled only for programs compiled with the /ANSI_STANDARD qualifier.

User Action: Change the argument to the TAB statement.

NETOPERR, network operation error (ERR=182)

Explanation: The program attempts to perform an invalid network operation, or the network software failed during a network operation.

User Action: Take action based on the associated error messages.

NODNAMERR, Node name error (ERR=175)

Explanation: A file specification's node name contains a syntax error.

User Action: Supply a valid node name.

NOTBASIC, Not a BASIC error (ERR=194)

Explanation: The error is not a HP BASIC error and is not mapped to an alternative HP BASIC error message.

User Action: Use RMSSTATUS or VMSSTATUS to access the text of the error message.

NOTENDFIL, Not at end of file (ERR=149)

Explanation: The program attempted a PUT operation: 1) on a sequential or relative file before the last record, or 2) without opening the file for WRITE access.

User Action: OPEN a sequential or relative file with ACCESS APPEND or OPEN the file with ACCESS WRITE.

NOTENOAVA, Not enough available memory (ERR=111)

Explanation: The program has exhausted its virtual space limits.

User Action: Raise the user PGFLQUOTA limit.

NOTENODAT, Not enough data in record (ERR=59)

Explanation: An INPUT statement did not find enough data in one line to satisfy all the specified variables.

User Action: Supply enough data in the record or reduce the number of specified variables.

NOTIMP, Not implemented (ERR=250)

Explanation: The program attempted to use a feature that does not exist in this version of HP BASIC, for example, TIME(4%).

User Action: Do not use the feature.

NOTRANACC, Not a random access device (ERR=64)

Explanation: The program attempts a random access on a device that does not allow such access, for example, a PUT with a record number to a magtape file.

User Action: Make the access sequential instead of random or use a suitable I/O device.

NO_CURREC, No current record (ERR=131)

Explanation: The program attempts a DELETE or UPDATE when the previous GET or FIND failed, or no previous GET or FIND was done.

User Action: Correct the cause of failure for the previous GET or FIND, or make sure a GET or FIND was done, then retry the operation.

NO_PRIKEY, No primary key specified (ERR=150)

Explanation: The program attempts to create an INDEXED file without specifying a PRIMARY KEY value.

User Action: Specify a PRIMARY KEY.

NO_ROOUSE, No room for user on device (ERR=4)

Explanation: No user storage space exists on the specified device.

User Action: Delete files that are no longer needed.

NUMCOOINS, Number of coordinates insufficient (ERR=281)

Explanation: Insufficient coordinates are provided. A GRAPH POINTS statement requires the coordinates for at least one point. A GRAPH LINES statement requires a minimum of two points. A GRAPH AREA statement requires a minimum of three points.

User Action: Supply an adequate number of points.

ONEOR_TWO, One or two dimensions only (ERR=102)

Explanation: The program contains a MAT statement that attempts to assign more than two dimensions to an array. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Change the number of dimensions in the MAT statement to one or two.

ON_STAOUT, ON statement out of range (ERR=58)

Explanation: The index value in an ON GOTO or ON GOSUB statement is less than one or greater than the number of line numbers in the list.

User Action: Check program logic to make sure that the index value is greater than or equal to one, and less than or equal to the number of line numbers in the ON GOTO or ON GOSUB statement.

OUTOF_DAT, Out of data (ERR=57)

Explanation: A READ statement requested additional data from an exhausted DATA list.

User Action: Remove the READ statement, reduce the number of variables in the READ statement, or supply more DATA items.

PRIKEYOUT, Primary key out of sequence (ERR=158)

Explanation: RMS has detected an error in a sequential PUT to an INDEXED file.

User Action: Change the PUT statement. If this does not work, the file is corrupted and you cannot do anything.

PRIUSIFOR, PRINT-USING format error (ERR=116)

Explanation: The program contains a PRINT USING statement with an invalid format string.

User Action: Change the PRINT USING format string.

PROC_TRA, Programmable ^C trap (ERR=28)

Explanation: A CTRL/C was typed at the controlling terminal.

User Action: None; however, you can trap this error with an error handler.

PROLOSSOR, Internal error in BASIC Run-Time Library. Please submit an SPR. (ERR=103)

Explanation: A consistency check in the HP BASIC run-time support failed. Program execution is aborted. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: This error should never occur. Submit a Software Performance Report.

PROVIO, Protection violation (ERR=10)

Explanation: The program attempted to read or write to a file whose protection code did not allow the operation.

User Action: Use a different file or change the file's protection code or the attempted operation.

RECALREXI, Record already exists (ERR=153)

Explanation: An attempted random access PUT on a relative file has encountered a pre-existing record.

User Action: Specify a different record number for the PUT or delete the record.

RECATTNOT, Record attributes not matched (ERR=228)

Explanation: A RECORDTYPE clause specifies record attributes that do not match those of the file.

User Action: Change the RECORDTYPE attribute to match that of the file.

RECBUCLOC, Record/bucket locked (ERR=154)

Explanation: The program attempts to access a record or bucket that has been locked by another program.

User Action: Retry the operation.

RECFILTOO, Record on file too big (ERR=157)

Explanation: The specified record is longer than the input buffer.

User Action: Increase the input buffer's size.

RECHASBEE, Record has been deleted (ERR=132)

Explanation: A record previously located by its Record File Address (RFA) has been deleted.

User Action: None.

RECNOTFOU, Record not found (ERR=155)

Explanation: A random access GET or FIND was attempted on a deleted or nonexistent record.

User Action: None.

RECNUMEXC, RECORD number exceeds maximum (ERR=147)

Explanation: The specified record number exceeds the maximum specified for this file.

User Action: Reduce the specified record number. The maximum record number cannot be specified in HP BASIC; it is either a default, or it was specified by a non-BASIC program when the file was created.

RECOVEMAP, RECORDSIZE overflows MAP buffer (ERR=185)

Explanation: The OPEN statement specifies a RECORDSIZE value larger than the size of the MAP specified in the MAP clause. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Increase the size of the MAP to match the RECORDSIZE value.

REDARR, Redimensioned array (ERR=105)

Explanation: A MAT statement attempts to redimension an array to have more elements than were originally dimensioned.

User Action: Change the statement that attempts the redimension or increase the original number elements.

REMOVEBUF, REMAP overflows buffer (ERR=183)

Explanation: A REMAP statement causes the variables in the dynamic MAP to be associated with nonexistent storage.

User Action: Change the REMAP statement so that all variables are associated with the storage in the MAP.

REMSTRNOT, REMAP string is not static (ERR=196)

Explanation: The program referenced a string with a REMAP statement that was not declared in COMMON or MAP.

User Action: Declare the string in the COMMON or MAP statement.

RESNO_ERR, RESUME and no error (ERR=104)

Explanation: The program executes a RESUME statement without a line number outside of the error handling routine. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Check program logic to make sure that the RESUME statement is executed only in the error handler.

RETWITGOS, RETURN without GOSUB (ERR=72)

Explanation: The program executes a RETURN statement before a GOSUB. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Check program logic to make sure that RETURN statements are executed only in subroutines or remove the RETURN statement.

RRVNOTFUL, RRV not fully updated, (ERR=171)

Explanation: RMS wrote a record successfully, but did not update one or more Record Retrieval Vectors; therefore, you cannot retrieve any records associated with those vectors.

User Action: Delete the record and rewrite it.

SCAFACINT, SCALE factor interlock (ERR=127)

Explanation: A subprogram was compiled with a different SCALE factor than that of the calling program. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Recompile one of the programs with a scale factor that matches the other.

SIZRECINV, Size of record invalid (ERR=156)

Explanation: The program contains a COUNT or RECORDSIZE specification that is invalid because:

- COUNT equals zero.
- COUNT exceeds the maximum size of the record.
- COUNT conflicts with the actual size of the current record during a sequential file UPDATE on disk.
- COUNT does not equal the record size for fixed format records.

- You specified a record size in the OPEN statement that was unequal to the actual record size established when the file was created.

User Action: Supply a valid COUNT value in the PUT or UPDATE statement, or a valid RECORDSIZE in the OPEN statement, whichever is applicable.

STO, Stop (ERR=123)

Explanation: The program executed a STOP statement. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = INFO or a greater severity.

User Action: Continue execution by typing CONTINUE or terminate execution by typing EXIT.

STRLENZER, string length is zero (ERR=288)

Explanation: A graphics statement references a null string where a null string is illegal.

User Action: Adjust the string length so that it is greater than zero.

STRTOOLON, String too long (ERR=227)

Explanation: The program attempts to create a string longer than 65,535 bytes.

User Action: Reduce the length of the string.

SUBOUTRAN, Subscript out of range (ERR=55)

Explanation: The program attempts to reference an array element outside of the array's dimensioned bounds.

User Action: Check program logic to make sure that all array references are to elements within the array boundaries.

TAPBOTDET, Tape BOT detected (ERR=129)

Explanation: The program attempts a rewind or backspace operation on a magnetic tape that is already at the beginning of the file.

User Action: Trap the error or check program logic; do not rewind or backspace if the magnetic tape is at the beginning of the file.

TAPNOTANS, Tape not ANSI labeled (ERR=146)

Explanation: The program attempts to access a file-structured magnetic tape that does not have an ANSI label.

User Action: Determine the magnetic tape's format by mounting it with the /FOREIGN qualifier and using the DCL DUMP command. You can then access it as a non-file-structured magnetic tape.

TAPRECNOT, Tape records not ANSI (ERR=128)

Explanation: The records in the magtape you accessed are neither ANSI D nor ANSI F format.

User Action: Determine the magtape's format by mounting it with the /FOREIGN qualifier and using the DCL DUMP command.

TERFORFIL, Terminal format file required (ERR=164)

Explanation: The program attempted to use PRINT #, INPUT #, LINPUT #, MAT INPUT #, MAT PRINT #, or PRINT USING # to access a RELATIVE, INDEXED, or VIRTUAL file.

User Action: Supply a terminal-format file.

TOOFEWARG, Too few arguments (ERR=97)

Explanation: A function invocation, CALL, or DRAW statement passed fewer arguments than were defined in the function, picture, DEF, DEF*, or subprogram. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Change the number of arguments to match the number defined in the function or subprogram.

TOOLITDAT, too little data in record (ERR=189)

Explanation: An INPUT statement did not find enough data in one line to satisfy all the specified variables. This error is signaled only for programs compiled with the /ANSI_STANDARD qualifier.

User Action: Supply enough data in the record, or reduce the number of specified variables.

TOOMANARG, Too many arguments (ERR=89)

Explanation: A function invocation, CALL, or DRAW statement passed more arguments than were expected. This error cannot be trapped with a HP BASIC error handler unless the program contains OPTION HANDLE = SEVERE.

User Action: Reduce the number of arguments. A SUB or FUNCTION subprogram can pass a maximum of 255 arguments; a DEF function call can pass a maximum of eight arguments.

TOOMUCDAT, too much data in record (ERR=177)

Explanation: The user has given too many items in response to the INPUT statement. This error is only signaled for ANSI INPUT.

User Action: Supply the correct number of items to the INPUT statement or change the INPUT statement.

TRANOTDIF, Transformation numbers are not different (ERR=260)

Explanation: The same transformation number is used twice in the SET INPUT PRIORITY statement.

User Action: Specify two different transformations in the SET INPUT PRIORITY statement.

UNEFILDAT, unexpired file date (ERR=179)

Explanation: The program attempts to delete a file whose expiration date has not yet passed.

User Action: None.

UNINUMNOT, Unit number is not defined for the device (ERR=282)

Explanation: The specified unit is a method that is not supported by the device. (The default unit is 1.)

User Action: Verify the supported units for the device and specify a valid unit.

UNKGKSERR, Unknown DEC GKS FOR VMS error (ERR=286)

Explanation: A graphics error has occurred that is not mapped to a HP BASIC error message.

User Action: Use VMSSTATUS to access the text of the Compaq GKS error message.

USEABOINP, User aborted input, locate point cancelled (ERR=293)

Explanation: The middle mouse button was pressed during the execution of a graphics input statement that uses a mouse to enter points (for example, LOCATE POINT). The pressing of the middle mouse button aborts the graphics input statement in progress and the data in the variables used for the graphics input statement is unchanged.

The pressing of the middle mouse button during a graphics input statement is analogous to typing Ctrl/Z at a regular INPUT statement.

User Action: None. The program can trap this error in an error handler and attempt the input statement again if so desired.

VIRARRDIS, Virtual array not on disk (ERR=43)

Explanation: The program attempted to reference a virtual array on a nondisk device, or the virtual array is not opened as ORGANIZATION VIRTUAL.

User Action: Virtual arrays must be on disk; change the file specification in the OPEN statement for this array. Open the file with ORGANIZATION VIRTUAL.

VIRARROPE, Virtual array not yet open (ERR=45)

Explanation: The program attempted to reference a virtual array before opening the associated disk file.

User Action: Open the disk file containing the virtual array before referencing the array.

VIRBUFTOO, Virtual buffer too large (ERR=42)

Explanation: The program attempted to access a VIRTUAL file and the buffer size was not 512 bytes.

User Action: Change the I/O buffer to be a multiple of 512 bytes.

B.2 HP BASIC Run-Time Errors by Number

Table B–1 shows the run-time errors by their number and gives an explanation of each error.

Table B–1 BASIC Run-Time Errors

Error Number	Explanation
1	BADDIRDEV, Bad directory for device
2	ILLFILNAM, Illegal file name
4	NO_ROOUSE, No room for user on device
5	CANFINFIL, Can't find file or account
7	IO_CHAALR, I/O channel already open
9	IO_CHANOT, I/O channel not open
10	PROVIO, Protection violation
11	ENDFILDEV, End of file on device
12	FATSYSIO_, Fatal system I/O failure
14	DEVHUNWRI, Device hung or write locked
15	KEYWAIEXH, Keyboard wait exhausted
18	ILLSYSUSA, Illegal SYS( ) usage
28	PROC_ _TRA, Programmable ^C trap
29	CORFILSTR, Corrupted file structure
31	ILLBYTCOU, Illegal byte count for I/O
35	MEMMANVIO, Memory management violation
42	VIRBUFTOO, Virtual buffer too large
43	VIRARRDIS, Virtual array not on disk
45	VIRARROPE, Virtual array not yet open
46	ILLIO_CHA, Illegal I/O channel
47	LINTOOLON, Line too long
48	FLOPOIERR, Floating point error or overflow
49	ARGTOOLAR, Argument too large in EXP
50	DATFORERR, Data format error
51	INTERR, Integer error
52	ILLNUM, Illegal number
53	ILLARGLOG, Illegal argument in LOG
54	IMASQUROO, Imaginary square roots

(continued on next page)

Table B–1 (Cont.) BASIC Run-Time Errors

Error Number	Explanation
55	SUBOUTRAN, Subscript out of range
56	CANINVMAT, Can't invert matrix
57	OUTOF_DAT, Out of data
58	ON_STAOUT, ON statement out of range
59	NOTENODAT, Not enough data in record
61	DIVBY_ZER, Division by 0
63	FIEOVEBUF, FIELD overflows buffer
64	NOTRANACC, Not a random access device
66	MISSPEFEA, Missing special feature
67	ILLSWIUSA, Illegal switch usage
72	RETWITGOS, RETURN without GOSUB
73	FNEWITFUN, FNEND without function call
88	ARGDONMAT, Arguments don't match
89	TOOMANARG, Too many arguments
97	TOOFEWARG, Too few arguments
101	DATTYPERR, Data type error
102	ONEOR_TWO, One or two dimensions only
103	PROLOSSOR, Internal error in BASIC Run-Time Library. Please submit an SPR.
104	RESNO_ERR, RESUME and no error
105	REDARR, Redimensioned array
116	PRIUSIFOR, PRINT-USING format error
122	ILLFIEVAR, Illegal FIELD variable
123	STO, Stop
124	MATDIMERR, Matrix dimension error
126	MAXMEMEXC, Maximum memory exceeded
127	SCAFACINT, SCALE factor interlock
128	TAPRECNOT, Tape records not ANSI

(continued on next page)

Table B–1 (Cont.) BASIC Run-Time Errors

Error Number	Explanation
129	TAPBOTDET, Tape BOT detected
130	KEYNOTCHA, Key not changeable
131	NO_CURREC, No current record
132	RECHASBEE, Record has been deleted
133	ILLUSADEV, Illegal usage for device
134	DUPKEYDET, Duplicate key detected
136	ILLILLACC, Illegal or illogical access
137	ILLKEYATT, Illegal key attributes
138	FILIS_LOC, File is locked
139	INVFILOPT, Invalid file options
141	ILLOPE, Illegal operation
142	ILLRECFIL, Illegal record on file
143	BADRECIDE, Bad record identifier
144	INVKEYREF, Invalid key of reference
145	KEYSIZTOO, Key size too large
146	TAPNOTANS, Tape not ANSI labelled
147	RECNUMEXC, RECORD number exceeds maximum
148	BADRECVAL, Bad RECORDSIZE value on OPEN
149	NOTENDFIL, Not at end of file
150	NO_PRIKEY, No primary key specified
151	KEYFIEBEY, Key field beyond end of record
152	ILLRECACC, Illogical record accessing
153	RECALREXI, Record already exists
154	RECBUCLOC, Record/bucket locked
155	RECNOTFOU, Record not found
156	SIZRECINV, Size of record invalid
157	RECFILTOO, Record on file too big
158	PRIKEYOUT, Primary key out of sequence

(continued on next page)

Table B–1 (Cont.) BASIC Run-Time Errors

Error Number	Explanation
159	KEYLARTHA, Key larger than record
160	FILATTNOT, File attributes not matched
161	MOVOVEBUF, Move overflows buffer
162	CANNOT OPEN FILE
164	TERFORFIL, Terminal format file required
166	NEGFILSTR, Negative fill or string length
168	ILLALLCLA, Illegal ALLOW clause
170	INDNOTFUL, Index not fully optimized
171	RRVNOTFUL, RRV not fully updated,
173	INVRFAFIE, Invalid RFA field
174	FILEXPDAT, File expiration date not yet reached
175	NODNAMERR, Node name error
176	NEGTABNOT, Negative TAB not allowed
177	TOOMUCDAT, Too much data in record
178	ERRFILCOR, Error on OPEN - file corrupted
179	UNEFILDAT, Unexpired file date
181	DECERR, Decimal error or overflow
182	NETOPERR, Network operation error
183	REMOVEBUF, REMAP overflows buffer
185	RECOVEMAP, RECORDSIZE overflows MAP buffer
186	IMPERRHAN, Improper error handling
187	ILLRECLOC, Illegal record locking
189	TOOLITDAT, Too little data in record
190	ILLNETOPE, Illegal network operation
191	ILLTFFOPE, Illegal terminal-format file operation
192	ILLWAIVAL, Illegal wait value
193	DEADLOCK, Detected deadlock while waiting for GET or FIND
194	NOTBASIC, Not a BASIC error

(continued on next page)

Table B–1 (Cont.) BASIC Run-Time Errors

Error Number	Explanation
195	DIMOUTRAN, Dimension number out of range
196	REMSTRNOT, REMAP string is not static
197	ARRTOOSMA, Array too small
226	GKSNOTINS, DEC GKS FOR VMS is not installed
227	STRTOOLON, String too long
228	RECATTNOT, Record attributes not matched
229	DIFUSELON, Differing use of LONG/WORD qualifiers
238	ARRMUSSAM, Arrays must be same dimension
239	ARRMUSSQU, Arrays must be square
240	CANCHAARR, Cannot change array dimensions
245	ILLEXIDEF, Illegal exit from DEF*
246	ERRTRANEE, ERROR trap needs RESUME
247	ILLRESSUB, Illegal RESUME to subroutine
250	NOTIMP, Not implemented
252	FILACPFAI, FILE ACP failure
253	DIRERR, Directive error
256	ECHTYPNOT, Prompt/echo type not supported
257	ILLTRANUM, Illegal transformation number
258	ILLPICOPE, Illegal picture operation
259	CLIPONOFF, Clipping must be ON or OFF
260	TRANOTDIF, Transformation numbers are not different
261	COLNOTCON, Color indices are not contiguous
262	ILLARESTY, Illegal area style
263	ILLTEXJUS, Illegal text justification
264	ILLTEXPRE, Illegal text precision
265	ILLTEXPAT, Illegal text path
266	ILLDEVID, Illegal device identification number
267	DEVTYPNOT, Device type is not supported

(continued on next page)

Table B–1 (Cont.) BASIC Run-Time Errors

Error Number	Explanation
268	DEVNOTOPE, Device is not open
269	DEVOUTMET, Device is an output metafile
270	DEVINMET, Device is an input metafile
272	DEVOPEING, Device and operation are incompatible
273	COONOTNDC, Coordinates are not within NDC space
274	ILLLINSTY, Illegal line style number
275	ILLLINSIZ, Illegal line size
276	ILLPOISTY, Illegal point style number
277	ILLTEXRAT, Illegal text width-to-height ratio
278	ILLTEXHEI, Illegal text height
279	ILLSTYIND, Illegal area style index
280	ILLCOLIND, Illegal color index
281	NUMCOOINS, Number of coordinates is insufficient
282	UNINUMNOT, Unit number is not defined for the device
283	ILLECHARE, Illegal echo area
284	ILLINIVAL, Illegal initial value
285	ENTPOINOT, Entered points not within a transformation
286	UNKGKSERR, Unknown DEC GKS FOR VMS error
287	INVCHASTR, Invalid character in string
288	STRLENZER, String length is zero
289	DATOVERF, Data overflow
290	ILLCNTCLA, Illegal count clause
291	ILLCOLMIX, Illegal color mix
292	ILLDEVNAM, Illegal device name in OPEN
293	USEABOINP, User aborted input, locate point cancelled

B.3 Errors Not Generated by HP BASIC

Table B–2 contains errors that cannot be generated in HP BASIC. However, they can be displayed with the ERT$ function and are included for completeness.

Table B–2 Errors Not Generated by HP BASIC

Number	Text
3	?Account or device in use
6	?Not a valid device
8	?Device not available
13	?User data error on device
16	?Name or account now exists
17	?Too many open files on unit
19	?Disk block is interlocked
20	?Pack ids don't match
21	?Disk pack is not mounted
22	?Disk pack is locked out
23	?Illegal cluster size
24	?Disk pack is private
25	?Disk pack needs 'cleaning'
26	?Fatal disk pack mount error
27	?I/O to detached keyboard
30	?Device not file-structured
32	?No buffer space available
33	?Odd address trap
34	?Reserved instruction trap
36	?SP stack overflow
37	?Disk error during swap
38	?Memory parity (or ECC) failure
39	?Magtape select error
40	?Magtape record length error

(continued on next page)

Table B–2 (Cont.) Errors Not Generated by HP BASIC

Number	Text
41	?Non-res run-time system
44	?Matrix or array too big
47	?Line too long
60	?Integer overflow, FOR loop
62	?No run-time system
65	?Illegal MAGTAPE( ) usage
68-70	unused
71	?Statement not found
74	?Undefined function called
75	?Illegal symbol
76	?Illegal verb
77	?Illegal expression
78	?Illegal mode mixing
79	?Illegal IF statement
80	?Illegal conditional clause
81	?Illegal function name
82	?Illegal dummy variable
83	?Illegal FN redefinition
84	?Illegal line number(s)
85	?Modifier error
86	?Can't compile statement
87	?Expression too complicated
90	%Inconsistent function usage
91	?Illegal DEF nesting
92	?FOR without NEXT
93	?NEXT without FOR
94	?DEF without FNEND
95	?FNEND without DEF
96	?Literal string needed

(continued on next page)

Table B–2 (Cont.) Errors Not Generated by HP BASIC

Number	Text
98	?Syntax error
99	?String is needed
100	?Number is needed
106	%Inconsistent subscript use
107	?ON statement needs GOTO
108	?End of statement not seen
109	?What?
110	?Bad line number pair
111	?Not enough available memory
112	?Execute only file
113	?Please use the run command
114	?Can't CONTinue
115	?File exists-RENAME/REPLACE
117	?Matrix or array without DIM
118	?Bad number in PRINT USING
119	?Illegal in immediate mode
120	?PRINT-USING buffer overflow
121	?Illegal statement
125	?Wrong math package
135	?Illegal usage
140	?Index not initialized
163	?No file name
165	?Cannot position to EOF
167	?Illegal record format
169	unused
172	?Record lock failed
180	?No support for operation in task
182	?Network operation rejected
184	?Unaligned REMAP variable

(continued on next page)

Table B–2 (Cont.) Errors Not Generated by HP BASIC

Number	Text
188	?UNLOCK EXPLICIT requires RECORDSIZE 512
198-225	unused
230	?No fields in image
231	?Illegal string image
232	?Null image
233	?Illegal numeric image
234	?Numeric image for string
235	?String image for numeric
236	?TIME limit exceeded
237	?First arg to SEG$ greater than second
241	?Floating overflow
242	?Floating underflow
243	?CHAIN to nonexistent line number
244	?Exponentiation error
248	?Illegal return from subroutine
249	?Argument out of bounds
251	?Recursive subroutine call
254-255	unused
294-300	unused

C

Optional Programming Productivity Tools

This appendix provides an overview of optional programming productivity tools. These tools are not included with the HP BASIC software; they must be purchased separately. Using these tools can increase your productivity as an HP BASIC programmer.

The following products are briefly described in this appendix:

- Digital Language Sensitive Editor for OpenVMS (LSE) and Digital Source Code Analyzer for OpenVMS (SCA) (Section C.1)
- Oracle CDD/Repository (Section C.2)
- Database Management System (DBMS) (Section C.3)
- Digital Test Manager for OpenVMS (Section C.4)
- Digital Code Management System for OpenVMS (CMS) (Section C.5)

For more information on using these tools, see the listed documentation at the end of each section.

For information about how to purchase these tools, contact your HP sales representative.

C.1 Language Sensitive Editor (LSE) and Source Code Analyzer (SCA)

The Digital Language Sensitive Editor for OpenVMS (LSE) and the Digital Source Code Analyzer for OpenVMS (SCA) must be purchased separately. LSE is a powerful and flexible text editor designed specifically for software development. LSE has important features that help you produce syntactically correct code in HP BASIC. SCA is an interactive tool that is used to perform program analysis.

LSE and SCA are closely integrated products; generally, SCA can be invoked through LSE. LSE provides additional editing features that make SCA program analysis more efficient. In addition, LSE and SCA, in conjunction with the HP BASIC compiler, provide a set of new enhancements supporting source code designing and review.

For more information about LSE and SCA, see the *Guide to Language-Sensitive Editor for VMS Systems* and *Guide to Source Code Analyzer for VMS Systems*.

C.1.1 Preparing an SCA Library

SCA stores data generated by the HP BASIC compiler in an SCA library. The data in the SCA library contains information about all symbols, modules, and files encountered during a specific compilation of the source.

After creating and initializing the OpenVMS directory of the SCA library, direct the HP BASIC compiler to generate data analysis files by appending the /ANALYSIS_DATA qualifier to the HP BASIC command as follows:

```
$ BASIC/ANALYSIS_DATA PG1,PG2,PG3
```

This command line compiles the input files PG1.BAS, PG2.BAS, and PG3.BAS, and generates corresponding output files for each input file with the file types OBJ and ANA. SCA puts these files in your current default directory.

Load the information in the data analysis files into your SCA library with the LOAD command as follows:

```
$ SCA LOAD PG1,PG2,PG3
```

This command loads your library with the modules contained in the data analysis files PG1.ANA, PG2.ANA, and PG3.ANA.

After the SCA library has been prepared, enter LSE to begin an SCA session. Within this context, the integration of LSE and SCA provides commands that can be used only within LSE.

C.1.2 Compiling From Within LSE

To compile a completed HP BASIC program, enter the following command at the LSE prompt:

```
LSE> COMPILE
```

To compile an HP BASIC program that contains placeholders and design comments, include the following qualifiers to the previous command:

```
LSE> COMPILE $/ANALYSIS_DATA/DESIGN=(PLACEHOLDERS, COMMENTS)
```

The /ANALYSIS_DATA qualifier causes the compiler to generate an analysis data file containing source code analysis information. This information is provided to the SCA library.

The /DESIGN qualifier instructs the HP BASIC compiler to recognize placeholders and design comments as valid program elements. If the /ANALYSIS_DATA qualifier has also been specified, the HP BASIC compiler includes information on placeholders and design comments in the analysis data file.

C.1.3 HP BASIC Support for LSE and SCA Features

This section describes information specific to BASIC for programming language placeholders and tokens.

LSE accepts keywords, or tokens, for all languages with LSE support. However, the specific tokens themselves are language defined. For example, you can expand the [MAT] token only when using HP BASIC.

Likewise, LSE provides placeholders, or prompt markers, for all languages with LSE support. However, as with tokens, the specific text or choices these markers call for are language defined. For example, you see the [record-declarations] placeholder only when using HP BASIC.

Note

Keywords such as TYPE, VARIANT, IF, FOR, and OPEN, can be tokens as well as placeholders. Therefore, any time you are in LSE with the language set to HP BASIC, you can type one of these words and press Ctrl/E to expand the construct.

Remember that braces ({}) enclose required placeholders and brackets ([]) enclose optional placeholders. Note that when you erase an optional placeholder, LSE also deletes any associated text before and after that placeholder.

You can use the SHOW TOKEN and SHOW PLACEHOLDER commands to display a list of all HP BASIC tokens and placeholders, or a particular token or placeholder. For example:

```
LSE> SHOW TOKEN IF              {lists the token IF}
LSE> SHOW TOKEN                 {lists all tokens}
```

To copy the listed information into a separate file, first enter the appropriate SHOW command to put the list into the $SHOW buffer. Next, enter the following command:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename.filetype
```

You can use the PRINT command to print the file you created.

C.2 CDD/Repository

HP BASIC supports CDD/Repository. The current version of CDD/Repository is compatible with previous versions of CDD.

See Chapter 21 for more information about CDD/Repository.

C.3 Database Management System (DBMS)

DBMS is a multiuser, general-purpose, CODASYL-compliant database management system. DBMS is used for accessing and administrating databases ranging in complexity from simple hierarchies to complex networks with multilevel relationships. DBMS supports full concurrent access in a multiuser environment without compromising the integrity and security of your database.

For more information, see the DBMS documentation.

C.4 Digital Test Manager for OpenVMS

The Test Manager helps test software during development and maintenance. This tool automates the organization, execution, and review of tests and allows several developers to use one set of tests at the same time.

With the Test Manager, you can describe your tests, organize them by assigning them to groups, and choose combinations of tests to run by test name or by group. The Test Manager executes the tests selected and then compares the result with the expected results.

For more information, see the *Guide to Test Manager for VMS Systems*.

C.5 Code Management System for OpenVMS (CMS)

The Code Management System for OpenVMS (CMS) is a program librarian for software development and evolution. It is comprised of a set of commands that enable you to manage files of an ongoing project.

For more information about CMS, see the *Guide to Source Code Analyzer for VMS Systems*.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z