BASIC on VAX/VMS Systems

Order No. AA–L336A–TK

November 1982

This manual teaches you how to get started with VAX–11 BASIC and explains how to use VAX/VMS features in a BASIC program.

OPERATING SYSTEMS AND VERSIONS:	VAX/VMS	V3
SOFTWARE VERSIONS:	VAX–11 BASIC	V2

digital equipment corporation, maynard, massachusetts

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

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

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

Copyright © 1982 by Digital Equipment Corporation. All Rights Reserved.

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

The following are trademarks of Digital Equipment Corporation:

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

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

Contents

Page

To the Reader — xi

Chapter 1 VAX–11 BASIC on VMS

1.1 Getting Started on VAX/VMS . . . 1–1
1.2 The BASIC Environment . . . 1–2
1.2.1 Creating and Running Programs . . . 1–3
1.2.2 Compiling Programs in the BASIC Environment . . . 1–4
1.2.3 Immediate Mode and Calculator Mode . . . 1–4
1.2.4 Debugging in Immediate Mode . . . 1–6
1.3 Using DCL to Perform Simple File Operations . . . 1–7
1.3.1 Creating Files . . . 1–8
1.3.2 Displaying Files . . . 1–9
1.3.3 Printing Files . . . 1–10
1.3.4 Deleting Files . . . 1–10
1.4 Using BASIC from DCL Command Level . . . 1–10
1.4.1 Compiling Programs Using the DCL BASIC Command . . . 1–11
1.4.2 Linking and Running Programs . . . 1–11

Chapter 2 Compiler Commands and Qualifiers

2.1 Using the BASIC Environment . . . 2–3
2.1.1 APPEND . . . 2–4
2.1.2 ASSIGN . . . 2–5
2.1.3 $ Command . . . 2–5
2.1.4 COMPILE . . . 2–5
2.1.5 CONTINUE . . . 2–7
2.1.6 DELETE . . . 2–8
2.1.7 EDIT . . . 2–8
2.1.8 EXIT . . . 2–9
2.1.9 HELP . . . 2–9
2.1.10 IDENTIFY . . . 2–11
2.1.11 LIST and LISTNH . . . 2–11
2.1.12 LOAD . . . 2–11
2.1.13 LOCK . . . 2–12
2.1.14 NEW . . . 2–12
2.1.15 OLD . . . 2–13
2.1.16 RENAME . . . 2–13
2.1.17 REPLACE . . . 2–13
2.1.18 RESEQUENCE . . . 2–14
2.1.19 RUN and RUNNH . . . 2–14

2.1.20 SAVE . . . 2–15
2.1.21 SCALE . . . 2–15
2.1.22 SCRATCH . . . 2–15
2.1.23 SEQUENCE . . . 2–15
2.1.24 SET . . . 2–16
2.1.25 SHOW . . . 2–16
2.1.26 UNSAVE . . . 2–18

2.2 Using BASIC from DCL Command Level . . . 2–19

2.2.1 ANSI_STANDARD . . . 2–21
2.2.2 AUDIT . . . 2–21
2.2.3 CHECK . . . 2–22
2.2.4 CROSS . . . 2–22
2.2.5 DEBUG . . . 2–23
2.2.6 DECIMAL_SIZE . . . 2–23
2.2.7 FLAG . . . 2–24
2.2.8 INTEGER_SIZE . . . 2–24
2.2.9 LINES . . . 2–24
2.2.10 LIST . . . 2–24
2.2.11 MACHINE . . . 2–25
2.2.12 OBJECT . . . 2–25
2.2.13 REAL_SIZE . . . 2–25
2.2.14 ROUND . . . 2–26
2.2.15 SCALE . . . 2–26
2.2.16 SHOW . . . 2–26
2.2.17 SYNTAX_CHECK . . . 2–27
2.2.18 TYPE_DEFAULT . . . 2–27
2.2.19 VARIANT . . . 2–27
2.2.20 WARNING . . . 2–27
2.2.21 Default Compiler Options . . . 2–28

2.3 Running and Debugging Multiple Program Units . . . 2–29

Chapter 3 Program Segmentation

3.1 Program Segmentation Techniques . . . 3–2
3.2 Declaring Subprograms . . . 3–3
3.3 Calling Subprograms . . . 3–4

3.3.1 Invoking Subprograms (CALL Statement) . . . 3–5
3.3.2 Parameter Passing Mechanisms . . . 3–6

3.3.2.1 Local Copies . . . 3–8
3.3.2.2 Argument Lists . . . 3–8

3.4 BASIC Subprograms . . . 3–8

3.4.1 SUB Subprograms . . . 3–9
3.4.2 Function Subprograms . . . 3–12

3.5 Compiling Subprograms . . . 3–13
3.6 Calling Non–BASIC Subprograms . . . 3–14
3.7 Calling BASIC Subprograms from Other Languages . . . 3–15

Chapter 4 The VAX–11 Symbolic Debugger

4.1 Debugger Symbol Table . . . 4–2

4.2 Preparing to Debug a Program 4–2

4.2.1 SET LANGUAGE and SHOW LANGUAGE Commands 4–3
4.2.2 SET MODULE, SHOW MODULE, and CANCEL MODULE Commands 4–3
4.2.3 SET SCOPE, SHOW SCOPE, and CANCEL SCOPE Commands 4–4

4.3 Controlling Program Execution 4–5

4.3.1 SET BREAK, SHOW BREAK, and CANCEL BREAK Commands 4–5
4.3.2 SET TRACE, SHOW TRACE, and CANCEL TRACE Commands 4–6
4.3.3 SET WATCH, SHOW WATCH, and CANCEL WATCH Commands 4–7
4.3.4 SHOW CALLS Command 4–8
4.3.5 GO and STEP Commands 4–8
4.3.6 CTRL/Y Command 4–10
4.3.7 EXIT Command 4–10

4.4 Examining and Modifying Locations 4–10

4.4.1 EXAMINE Command 4–10
4.4.2 DEPOSIT Command 4–11
4.4.3 EVALUATE Command 4–12
4.4.4 Specifying Address 4–12

4.4.4.1 Specifying Scope 4–13
4.4.4.2 Previous, Current, and Next Locations 4–13
4.4.4.3 Defining Addresses Symbolically 4–14

4.4.5 Calling Subroutines from the Debugger 4–14
4.4.6 Debugger Command Qualifiers 4–14
4.4.7 Numeric Data Types 4–15
4.4.8 Evaluating Named Constants 4–15

Chapter 5 System Services and Run-Time Library Procedures

5.1 System Services 5–2

5.1.1 VAX/VMS Symbolic Constants 5–2
5.1.2 Testing for Success or Failure (System Status Codes) 5–2
5.1.3 Declaring System Services and Symbolic Status Codes: EXTERNAL Statement 5–4
5.1.4 System Service Examples 5–4

5.1.4.1 Creating a Mailbox: SYS$CREMBX 5–5
5.1.4.2 Translating a Logical Name: SYS$TRNLOG 5–6
5.1.4.3 Translating System Status Codes: SYS$GETMSG 5–7
5.1.4.4 Queueing I/O Requests: SYS$QIOW 5–10
5.1.4.5 Getting Information About a Job/Process: SYS$GETJPI 5–12

5.1.5 Resolving External Names 5–14

5.2 Calling Run-Time Library Routines 5–14

5.2.1 Types of RTL Procedures 5–15
5.2.2 Measuring Performance: LIB$INIT_TIMER, LIB$FREE_TIMER, and LIB$STAT_TIMER 5–16
5.2.3 Using Logical Unit Numbers: LIB$GET_LUN and LIB$FREE_LUN 5–18
5.2.4 Using Arccosine Procedures: MTH$ACOS 5–20

Chapter 6 The RECORD Statement

6.1 The RECORD Statement . . . 6–1
6.1.1 Grouping RECORD Components . . . 6–2
6.1.2 RECORD Variants . . . 6–5
6.1.3 Accessing RECORD Components . . . 6–6

Chapter 7 ANSI Minimal BASIC

7.1 Introduction . . . 7–1
7.2 /ANSI_STANDARD Qualifier . . . 7–1
7.3 Extensions to X3.60–1978 . . . 7–2
7.3.1 Statements . . . 7–2
7.3.2 Variables . . . 7–2
7.3.3 Numeric Constants . . . 7–2
7.3.4 User-Defined Functions (DEF Statement) . . . 7–3
7.3.5 Built-In Functions . . . 7–3
7.3.6 Arrays . . . 7–4
7.3.7 Program Format . . . 7–4
7.4 Implementation-Defined Features . . . 7–4

Chapter 8 I/O on VAX/VMS

8.1 Using Logical Names in File Specifications . . . 8–2
8.2 RMS I/O to Magnetic Tape . . . 8–2
8.2.1 Allocating and Mounting the Tape . . . 8–2
8.2.2 Opening FOR OUTPUT . . . 8–3
8.2.3 Opening FOR INPUT . . . 8–3
8.2.4 Positioning the Tape (NOREWIND) . . . 8–4
8.2.5 Accessing ANSI Magnetic Tape Files . . . 8–4
8.2.5.1 Writing Records (PUT) . . . 8–4
8.2.5.2 Reading Records (GET) . . . 8–5
8.2.6 Speeding Access Time (BLOCKSIZE) . . . 8–5
8.2.7 Blocking Records . . . 8–6
8.2.8 Rewinding the Tape (RESTORE) . . . 8–6
8.2.9 Closing the File (CLOSE) . . . 8–6
8.3 Device-Specific I/O . . . 8–7
8.3.1 Device-Specific I/O to Unit Record Devices . . . 8–7
8.3.2 Device-Specific I/O to Magnetic Tape Devices . . . 8–7
8.3.2.1 Allocating and Mounting the Tape . . . 8–7
8.3.2.2 Opening FOR OUTPUT . . . 8–8
8.3.2.3 Opening FOR INPUT . . . 8–8
8.3.2.4 Writing Records (PUT) . . . 8–9
8.3.2.5 Reading Records (GET) . . . 8–9
8.3.2.6 Rewinding the Tape (RESTORE) . . . 8–10
8.3.2.7 Closing the Tape (CLOSE) . . . 8–10

8.3.3 Device-Specific I/O to Disks . . . 8–10
8.3.3.1 Assigning and Mounting the Disk . . . 8–10
8.3.3.2 Opening for OUTPUT . . . 8–11
8.3.3.3 Opening for INPUT . . . 8–11
8.3.3.4 Accessing the Disk . . . 8–11
8.3.3.5 Writing Records (PUT) . . . 8–12
8.3.3.6 Reading Records (GET) . . . 8–12
8.4 Input and Output to Mailboxes . . . 8–13
8.5 Network I/O . . . 8–14
8.5.1 Remote File Access . . . 8–14
8.5.2 Task-to-Task Communication . . . 8–14
8.6 File Sharing and Explicit Record Locking . . . 8–16
8.6.1 Explicit Record Locking . . . 8–16
8.7 Sample Programs . . . 8–18

Chapter 9 The VAX–11 Common Data Dictionary

9.1 Overview . . . 9–1
9.2 The VAX–11 Common Data Dictionary (CDD) . . . 9–1
9.2.1 Character String Data Types . . . 9–6
9.2.2 Integer (Fixed-Point) Data Types . . . 9–7
9.2.3 Floating-Point Data Types . . . 9–9
9.2.4 Decimal String Data Types . . . 9–11
9.2.5 Other Data Types . . . 9–12
9.2.6 Arrays . . . 9–13

Chapter 10 Object Module Libraries and Shareable Images

10.1 User Supplied Libraries . . . 10–1
10.1.1 Creating Libraries . . . 10–1
10.1.2 Accessing User Supplied Libraries . . . 10–2
10.2 The System Library . . . 10–3
10.3 Shareable Images . . . 10–4

Appendix A Compile-Time Error Messages

A.1 Compile-Time Errors . . . A–1
A.2 BASIC Environment Errors . . . A–43
A.3 Shared Error Messages . . . A–47
A.4 Informational Messages from Flaggers . . . A–48

Appendix B Run-Time Error Messages

B.1 VAX–11 BASIC Run-Time Errors by Mnemonic . . . B–1
B.2 VAX–11 BASIC Run-Time Errors by Number . . . B–18
B.3 Errors Not Generated by VAX–11 BASIC . . . B–21

Appendix C ASCII Codes and Data Representation

C.1 ASCII Character Codes C–1
C.2 Integer Format. C–4
C.2.1 Byte-Length Integer Format C–4
C.2.2 Word-Length Integer Format C–5
C.2.3 Longword Integer Format. C–5
C.3 Real Number Formats C–5
C.3.1 SINGLE Floating-Point Number Format (F_floating) C–5
C.3.2 DOUBLE Floating-Point Number Format (D_floating) C–6
C.3.3 GFLOAT Floating-Point Number Format (G_floating). C–6
C.3.4 HFLOAT Floating-Point Number Format (H_floating). C–7
C.4 Packed Decimal Number Format C–7
C.5 String and Array Descriptor Format. C–8
C.5.1 Fixed-Length String Descriptor Format C–8
C.5.2 Dynamic String Descriptor Format C–8
C.6 Array Descriptors C–8
C.7 Decimal Scalar String Descriptor (Packed Decimal String Descriptor) C–10

Appendix D Example Programs

D.1 VAX–11 BASIC Record Sort. D–1
D.2 VAX–11 BASIC USEROPEN Routine D–14

Appendix E Listing File Format

Index

Figures

2–1 Program Development Methods 2–2
2–2 Running Multiple Program Units. 2–29
C–1 Byte-Length Integer Format C–4
C–2 Word-Length Integer Format C–5
C–3 Longword Integer Format C–5
C–4 Single-Precision Real Number Format C–6
C–5 Double-Precision Real Number Format C–6
C–6 Fixed-Length String Descriptor Format C–8
C–7 Dynamic String Descriptor Format C–8
C–8 Array Descriptor Format C–9
C–9 Decimal Scalar String Descriptor. C–10

Tables

1–1 File Specification Defaults 1–8
2–1 BASIC Compiler Commands 2–3
2–2 COMPILE Command Qualifiers 2–6
2–3 Qualifiers of the DCL BASIC Command. 2–20
3–1 Allowable Parameter Passing Mechanisms 3–6
4–1 Debugger Commands and Keywords 4–2
4–2 Debugger Command Qualifiers 4–15
5–1 System Service Status Codes 5–3
9–1 Data Types Common to the CDD and BASIC 9–4
9–2 CDD Data Types and BASIC Translation 9–4
C–1 ASCII Character Codes C–1

To the Reader

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

For the beginner:

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

For all systems:

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

For specific systems:

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

For the system manager:

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

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

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

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

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

Document Objectives

This manual describes the features and use of the BASIC language on VAX/VMS systems. It should be used with the *BASIC Reference Manual* and the *BASIC User's Guide*.

Intended Audience

This manual has two intended audiences. The first chapter is intended for programmers unfamiliar with both BASIC and VAX/VMS. This chapter teaches you how to get started with BASIC on VAX/VMS systems.

The remaining chapters are intended for programmers familiar with computer concepts, the BASIC language, and VAX/VMS systems. These chapters teach you how BASIC interacts with the VAX/VMS operating system and with other layered software.

This manual also contains the appendixes explaining the BASIC compile-time and run-time error messages.

Document Structure

This manual has 10 chapters and 5 appendixes.

Chapter 1	Introduces you to VAX–11 BASIC on the VAX/VMS operating system.
Chapter 2	Describes the BASIC compiler commands and their qualifiers, and the DCL BASIC command and its qualifiers.
Chapter 3	Describes how to create and use BASIC subprograms.
Chapter 4	Describes how to use the VAX–11 Symbolic Debugger to debug BASIC programs.
Chapter 5	Describes System Services and Run-Time Library routines.
Chapter 6	Explains how to create user-defined data structures with the RECORD statement.
Chapter 7	Explains how to write programs conforming to the ANSI Minimal BASIC Standard.
Chapter 8	Describes BASIC I/O on VAX/VMS systems.
Chapter 9	Explains how definitions extracted from the VAX–11 Common Data Dictionary are translated to BASIC RECORD statements.
Chapter 10	Describes the use of user-supplied libraries and shareable images.
Appendix A	Lists compile-time error messages.
Appendix B	Lists run-time error messages.
Appendix C	Lists ASCII Codes and describes data representation.
Appendix D	Contains example programs.
Appendix E	Explains the format of program listings.

Conventions

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

Syntax formats consist of BASIC keywords, metalanguage mnemonics, and punctuation symbols. Meta language mnemonics are symbolic derivations of BASIC objects or structures.

Note

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

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

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

Others are abbreviated forms of words. For example:

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

Most mnemonics used in formats are combinations of mnemonics:

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

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

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

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

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

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

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

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

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

When you look at this format, you can see that a data-type keyword must follow the DECLARE statement and that a *decl-item* must follow the data-type keyword. *Decl-item* is a complex mnemonic that is further defined in the lower portion of the box, like this:

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

From this portion of the format, you can see that the data-type keyword must be followed by an array name or a simple variable name. The portion of the format in brackets indicates that you can specify another data-type keyword and another array name or simple variable name. The comma and ellipsis (...), as defined on the tabbed divider in this section, indicates that you can continue adding data-type keywords and array names or simple variable names.

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

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

The most frequently used punctuation symbols and metalanguage mnemonics are listed and described in the following two tables. Less frequently used mnemonics and most complex mnemonics are defined as they occur in syntax formats.

Mnemonic	Definition
exp	An expression
vbl	A variable
unsubs	Unsubscripted; used with the variable mnemonic to indicate a simple variable, as opposed to an array element
subs	Subscripted; used with the variable mnemonic to indicate an array element; the element's position in the array is specified by subscripts enclosed in parentheses and separated by commas
array	An array; syntax formats indicate whether you can specify bounds and dimensions, or just dimensions
const	A constant value
lit	A literal value, in quotation marks; a literal is always a constant, but a constant may be named, so constants are not always literals
num	A numeric value
real	A floating-point value
int	An integer value
str	A character string
cond	Conditional; used with the expression mnemonic to indicate that an expression can be either logical or relational
log	Logical; used with the expression mnemonic to indicate a logical expression
rel	Relational; used with the expression mnemonic to indicate a relational expression
lex	Lexical; used to indicate a component of a compiler directive
target	The target point of a branch statement; used to indicate that the target point can be either a program line number or a statement label
lin-num	A program line number
label	An alphanumeric statement label
item	Allowable BASIC objects, such as variables, data types, and parameters; allowable objects are defined in formats as they occur
nam	Name; indicates the declaration of a name or the name of a BASIC structure, such as a SUB subprogram
com	Specific to a COMMON
def	Specific to a DEF
func	Specific to a FUNCTION subprogram
map	Specific to a MAP
sub	Specific to a SUB subprogram
chnl	An I/O channel associated with a file
data-type	A data-type keyword; Table 1–2 in this manual lists and describes BASIC data-type keywords
file-spec	A file-specification
file-nam	A file name

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

Definitions

In this manual, the following definitions apply:

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

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

Chapter 1
VAX–11 BASIC on VMS

This chapter introduces you to BASIC on VAX/VMS systems. It teaches you how to log in and out of the system and also explains how to make the VAX/VMS operating system perform simple tasks like printing and displaying files. If you are a newcomer to computer programming, you may want to read some of the introductory material for BASIC before reading this manual. These introductory manuals are:

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

When you write programs in BASIC, you can choose between two program development methods. You can write your source program with a text editor, then compile, link and run the program with commands to the VAX/VMS operating system (these are called DCL commands; DCL is an acronym for the Digital Command Language). This method is common to most programming languages. Alternatively, you can enter the BASIC environment, type in your program, and execute it with the BASIC RUN command.

The BASIC environment is especially useful for beginners but has many advanced features for the expert user. As you gain experience, you will probably use both the BASIC environment and DCL commands to develop BASIC programs

Note

The examples in this chapter use red ink to show characters typed by the user.

1.1 Getting Started on VAX/VMS

The VAX/VMS operating system controls the operation of the computer hardware and the sharing of computer resources among system users. If you are unfamiliar with the VAX–11 computer, you should read the *VAX/VMS Primer*.

This section describes, but does not completely explain, some of the DCL commands that let you manipulate files. For a complete explanation of these commands, see the *VAX/VMS Command Language User's Guide*.

The process of accessing the computer from a terminal is called "logging in." During this procedure, you identify yourself to the computer by typing your username and password. However, before you can log in, you must be authorized to use the system. That is, you must have an account. Accounts are set up by the system manager, or whoever is responsible at your installation for authorizing the use of the system. This person must provide you with your username and password.

Your username is a unique name that distinguishes you from other users. In many cases, a username is the same as a person's real first or last name. Your password is a unique word that confirms your identity, helping prevent other users from accessing your files.

When you type your username, you see it displayed on the terminal. However, when you type your password, you do not see it displayed because the computer does not "echo" it. This makes it harder for someone to see your password.

You begin to log in by typing the RETURN key. The system then prompts you for your username and password. For example:

```
Username: SMITH
Password:
        Welcome to VAX/VMS Version 3
$
```

If you correctly type your username and password, the system prints an identification message and prompts with a dollar sign. The dollar sign tells you that the system is ready to receive DCL commands.

Ending your session with the computer is called "logging out." You do this with the LOGOUT command:

```
$ LOGOUT
```

1.2 The BASIC Environment

In the BASIC environment, you type in numbered program lines, ending each line by typing the RETURN key. BASIC stores the source statements in memory in ascending line number sequence. You use compiler commands to modify, save, and execute the source programs. BASIC can also be used as a traditional compiler. See Section 1.4 for more information on using BASIC from DCL command level.

If you type BASIC and then the RETURN key, you enter the BASIC environment. For example:

```
$ BASIC(RET)
```

When you type this command, the operating system places you in the BASIC environment. BASIC prints an identification line displaying the software version and prompts with "Ready:"

```
Ready
```

You can respond with: 1) BASIC program lines, 2) compiler commands and qualifiers, or 3) immediate mode statements.

When you type program statements, BASIC stores them as part of the current program in memory. If you enter a program line with the same line number as an existing program line, the new line replaces the old one.

If a program line is too long for one text line, you can continue it by entering an ampersand (&) as the last character before typing the RETURN key. See Chapter 2 for more information about using BASIC compiler commands. See Section 1.3.3 for more information about immediate mode statements.

1.2.1 Creating and Running Programs

Once you have typed in your program, you can execute it with the BASIC RUN command. Or, you can save the program in a disk file with the SAVE command.

The following example shows a simple program being entered and run in the BASIC environment.

```
$ BASIC(RET)
VAX-11 BASIC V2.0
Ready

NEW FIRSTTRY(RET)

Ready

100     PRINT 'This program displays the product of two numbers.'(RET)
200     INPUT 'Numbers'; A, B(RET)
300     PRINT 'Their product is '; A * B(RET)
400     END(RET)
```

1. `$ BASIC`

   This command invokes BASIC. BASIC prints an identification header and prompts with "Ready."

2. `NEW FIRSTTRY`

   This command tells BASIC that the following program is new, and assigns it the name FIRSTTRY. BASIC again prompts with "Ready."

3. At this point you type in the four BASIC statements that make up this program.

Now BASIC is ready to accept more commands or more program lines. To execute the program, type:

```
RUN(RET)

FIRSTTRY                    27-JUL-1981  10:27

This program displays the product of two numbers.
Numbers? 5,6(RET)
Their product is 30
Ready
SAVE(RET)
```

1. The RUN command causes BASIC to identify the program, print the date and time, and begin executing the program. The program prints its message and prompts for the numbers to be multiplied. You type in numbers 5 and 6.

2. The program prints the product and ends execution.

3. BASIC again prompts with "Ready."

4. The SAVE command causes BASIC to save a copy of the program with the file specification FIRSTTRY.BAS (since you did not specify a file type, BASIC uses the default: BAS).

1.2.2 Compiling Programs in the BASIC Environment

Once you are satisfied with the way a program runs in the BASIC environment, you are ready to create an executable image of the program.

Running an executable image is more efficient than running the same program in the BASIC environment. Each time you read a program into memory with the OLD command and execute the program with the BASIC RUN command, BASIC must compile, link, and execute the program. When you create an executable image, the compile and link operations are done only once.

The first step in creating an executable image is to compile the program. This creates an object module file that the linker uses to create the executable image. To compile a program in the BASIC environment, type:

```
Ready
OLD FIRSTTRY(RET)
Ready
COMPILE(RET)
Ready
```

1. The OLD command tells BASIC to read a program called FIRSTTRY.BAS into memory.

2. The COMPILE command tells BASIC to create a linkable object file from the source program. The object file has the same file name as the source file (FIRSTTRY) and a file type of OBJ.

You can specify many options when compiling a program. For example, you may want to create a line printer listing of the program and the compilation. To do this, you use the /LIST qualifier to the COMPILE command:

```
COMPILE/LIST(RET)
```

You might also want to specify that floating-point numbers be more precise. To do this you specify:

```
COMPILE/DOUBLE(RET)
```

See Chapter 2 for a complete list of compiler commands and qualifiers.

After you have created the object module, you are ready to create an executable image of the program. To do this, you use the VAX–11 Linker, invoked by the DCL LINK command. See Section 1.4.2 for more information about linking programs.

1.2.3 Immediate Mode and Calculator Mode

You need not write a complete program to use VAX–11 BASIC. Many statements are executable in *immediate mode* or *calculator mode*.

Immediate and calculator mode statements are BASIC statements that are not preceded a line number. When you end such a statement by typing the RETURN key, BASIC executes it immediately. The difference between immediate mode and calculator mode is that immediate mode statements are entered after a program executes a STOP statement.

Thus with immediate mode statements, you can examine and change variables in the current program or perform calculations independent of the current program.

Calculator mode statements can be entered at any time. They let you use the BASIC environment as a desk calculator. Both types of statements must fit on a single line.

For example, because it contains a line number, BASIC stores this statement as part of the program in memory:

```
10 PRINT 'THIS IS AN EXECUTABLE BASIC STATEMENT'
```

However, if you type:

```
PRINT 'THIS IS AN IMMEDIATE MODE STATEMENT'
```

BASIC displays:

```
THIS IS AN IMMEDIATE MODE STATEMENT

Ready
```

The Ready prompt indicates that BASIC is ready to receive commands, more immediate mode statements, or new program lines.

Unless BASIC has executed a STOP statement in the current program, immediate mode statements are said to be done in calculator mode. This means that BASIC compiles and executes each immediate mode statement as if it were a self-contained program. For example:

```
PRINT PI * 67.3
 211.421
```

Each calculator mode program line must fit on a single text line. However, you can have more than one BASIC statement on a line if you separate them with a backslash (\):

```
A = (54.37 / 1.25) \ B = (328.15 ^ 2) \ PRINT (B / A)
 2475.69
```

Even if the current program has executed a STOP statement, you can still perform independent calculations. However, you should understand that after a STOP, any immediate mode statement referencing program variables uses the values assigned in the program.

If the current program has not executed a STOP statement, each immediate mode line exists by itself, and any data used by the statements on the line is temporary. For example:

```
A = 2 ^ 5 \ PRINT A
 32
Ready
PRINT A
 0
```

The second time that "PRINT A" is typed, BASIC displays a zero. This is because BASIC treats A as a new variable, and initializes it to the value zero.

You can use the IF, WHILE, UNTIL, UNLESS, and FOR statement modifiers in calculator mode statements. (Refer to Chapter 3 in the *BASIC User's Guide* for information on statement modifiers.) Thus, you can generate a table of square roots by typing:

```
PRINT I, SQR(I) FOR I = 1 TO 10

 1              1
 2              1.41421
 3              1.73205
 4              2
 5              2.23607
 6              2.44949
 7              2.64575
 8              2.82843
 9              3
 10             3.16228

Ready
```

Some statements are invalid in immediate mode. In general, these are the statements that require the allocation of new storage or statements that make no sense in the context of a single line. Invalid immediate mode statements include:

COMMON	DATA	DEF
DECLARE	END	SUB
FUNCTION	MAP	EXIT
ON	ONERROR	RESUME
RETURN		

If you try to execute such a statement, BASIC prints "Illegal in immediate mode."

1.2.4 Debugging in Immediate Mode

Debugging is the process of finding logic errors in a program. Even though a program compiles without errors, it may not execute the way you expect.

Immediate mode debugging is done in the context of a BASIC program, rather than the context of a single line. This feature lets you halt program execution with a STOP statement, examine and modify program data, then continue execution with the CONTINUE command. However, when immediate mode statements execute in the context of a program, you cannot create new variables; you can only examine and modify existing program variables. For example:

```
Ready

OLD BALANCE

LIST

BALANCE

100     PRINT 'This program keeps track of your checking balance.'
200     INPUT 'Starting balance'; BALANCE
300     INPUT 'How many checks'; NUM_CHECKS%
400     FOR I% = 1% TO NUM_CHECKS%
500             INPUT 'Amount of check'; CHECK_AMOUNT
600             BALANCE = BALANCE - CHECK_AMOUT
700     NEXT I%
800     PRINT 'Your balance is '; BALANCE
```

This program executes with no error messages. However, no matter how many checks you write, the balance always remains at the starting value you type in. To find out where the problem lies, you might insert a STOP statement after line 600:

```
601 STOP
RUNNH

This program keeps track of your checking balance.
Starting balance? 500.00(RET)
How many checks? 3(RET)
Amount of check? 12.50(RET)
%BAS-I-STO, Stop
-BAS-I-FROLINMOD, from line 601 in module BALANCE
PRINT BALANCE
 500
Ready
PRINT CHECK_AMOUNT
 12.5
Ready
PRINT BALANCE - CHECK_AMOUNT
 487.5
```

The program stops after executing line 600, and you can examine the program variables BALANCE and CHECK_AMOUNT and perform subtraction using these variables. However, the BALANCE variable still contains the starting balance. You now take a closer look at the program line performing the calculation:

```
LIST 600

600                    BALANCE = BALANCE - CHECK_AMOUT
```

You notice that the variable CHECK_AMOUNT is misspelled in line 600. This means that BASIC subtracted a variable named CHECK_AMOUT from the variable BALANCE. Because BASIC variables are initialized to zero, line 600 did not affect the balance at all. To correct the program, type:

```
600                    BALANCE = BALANCE - CHECK_AMOUNT

REPLACE
```

By typing in a new line 600, you replace the old one with the misspelled variable. The REPLACE command replaces the incorrect program with the current correct program.

1.3 Using DCL to Perform Simple File Operations

Almost all operations on the computer involve files. A file always has a name or specification. A file specification often indicates the input file to be processed or the output file to be produced. You need not give complete file specifications each time you compile, link, or execute a program. Usually, the only required part is the file name. File specifications have the form:

node::device:[directory]filename.filetype;version

where:

node Specifies a network node name or, if more than one is specified, a network node name path. Nodes apply only to systems that support VAX–11 DECnet.

device Identifies the hardware device on which a file is stored or is to be written.

directory Identifies the name of the directory under which the file is catalogued on the specified device. (You can delimit the directory name with either square brackets, as shown, or angle brackets (< >).)

filename Identifies the file by its name; filename can be up to nine characters long.

filetype Describes the kind of data in the file; filetype can be up to three characters long.

version Specifies the file's version number. Versions are identified by a decimal number, which is incremented by one each time you create a new version of a file. Either a semicolon or a period can separate filetype and version.

This is an example of a complete file specification:

```
BOSTON::DBA2:[JONES]PROG.BAS;4
```

Often the file name is the only required part of a file specification. If you omit any other part, a default value is used as shown in Table 1–1.

Table 1–1: File Specification Defaults

Optional Element	Default Value	
node	Local network node	
device	User's current default device	
directory	User's current default directory	
filetype	One of the following:	
	Input to compiler	BAS
	Output from compiler	OBJ
	Input to linker	OBJ
	Output from linker	EXE
	Input to DCL RUN command	EXE
	Compiler source listing	LIS
	Linker MAP listing	MAP
version	Input: highest existing version Output: highest existing version plus one	

The following sections describe some DCL commands that involve files.

1.3.1 Creating Files

To create text files you use a text editor. The DIGITAL standard text editor is called EDT and runs on many DIGITAL computers and operating systems. To invoke EDT on a VAX/VMS system, type:

```
$ EDIT file-spec(RET)
```

If the file you specify already exists, you can modify it with EDT. If the file does not exist, EDT prints:

```
Input file does not exist
```

In either case, EDT prompts with an asterisk for the next command. You can use EDT whether you have a hardcopy (for example, an LA36 or LA120) or video terminal (for example, a VT52 or VT100). The following is a sample EDT terminal session using LINE mode, which works on both hardcopy and video terminals.

```
$ EDIT/EDT FIRSTTRY.BAS(RET)
Input file does not exist
*INSERT(RET)
   100  PRINT 'This program displays the product of two numbers.'(RET)
   200  INPUT 'Numbers'; A, B(RET)
   300  PRINT 'Their product is '; A * B(RET)
   400  END(RET)
   ^Z
*EXIT(RET)
_DBA1:[JONES]FIRSTTRY.BAS;1 4 lines
$
```

1. This example first invokes the EDT editor, specifying that the file to be created is FIRSTTRY.BAS. EDT informs you that this is a new file, and the asterisk prompt tells you that EDT is in LINE mode.

2. The INSERT command tells EDT to insert all subsequent text until a CTRL/Z is typed.

3. You type in your BASIC program. When all lines have been entered, a CTRL/Z is typed. EDT then prompts for another command.

4. The EXIT command tells EDT to store the text you typed (the contents of the current buffer) in the file. EDT responds with the file specification of the created file: disk DBA1:, account [JONES], file FIRSTTRY.BAS;1. This is the file specified when EDT was invoked. EDT exits and returns your terminal to DCL command level.

This has been a very simple example of EDT. EDT has many other capabilities and features. See the *EDT Reference Manual* for more information.

1.3.2 Displaying Files

To find out what files are in your account, use the DCL DIRECTORY command. At its simplest, DIRECTORY displays the names, types, and versions of all the files in your account or "directory." For example:

```
$ DIRECTORY(RET)

FIRSTTRY.BAS;1     LOGIN.COM;1       MYPROG.BAS;1     MYPROG1.BAS;1

TEST.TXT;1

Total of 5 files.
```

You can display the contents of a file with the DCL TYPE command. For example:

```
$ TYPE FIRSTTRY.BAS(RET)
100     PRINT 'This program displays the product of two numbers.'
200     INPUT 'Numbers'; A, B
300     PRINT 'Their product is '; A * B
400     END
```

1.3.3 Printing Files

To print the file on the line printer, use the DCL PRINT command:

```
$ PRINT FIRSTTRY.BAS<RET>
Job 442 entered on queue SYS$PRINT
```

The message tells you that the file has been submitted to the print queue. To display the print queue, use the DCL SHOW command:

```
$ SHOW QUEUE SYS$PRINT<RET>
  * Generic Device queue "SYS$PRINT"  Burst Flag
    Job   440 JONES        ZULU    , Pri=4
    Job   442 SMITH        FIRSTTRY  , Pri=4
```

The response tells you that your request is awaiting processing in the print queue, SYS$PRINT. The listing will be printed in its turn.

1.3.4 Deleting Files

To remove unwanted files from your directory, use the DCL DELETE command:

```
$ DELETE FIRSTTRY.BAS;1<RET>
```

You must include either a specific version number or an asterisk in the version number position. If you do not, the system prints:

```
%DELETE-E-DELVER, explicit version number or wildcard required
```

The asterisk specifies that you want to delete all files with the specified file name and file type. This version number requirement helps prevent you from deleting a file by mistake.

1.4 Using BASIC from DCL Command Level

The DCL BASIC command invokes the BASIC compiler. If you type BASIC followed by the RETURN key in response to the dollar sign prompt, you enter the BASIC environment. For example:

```
$BASIC<RET>
```

However, if you type BASIC followed by a file specification, BASIC tries to compile the program in the specified file. For example:

```
$ BASIC file-spec<RET>
```

This file may have been created with an editor, or it may have been entered in the BASIC environment, then written to a disk file with the SAVE or REPLACE commands. If the compilation succeeds, BASIC creates an object module with the same name as the source file, and a file type of OBJ. This object file is used by the VAX–11 Linker to create an executable image. The following sections describe these operations in more detail.

1.4.1 Compiling Programs Using the DCL BASIC Command

To compile a source program at DCL command level, use the BASIC command. Its format is:

$ BASIC[/qualifier(s)] file-spec-list[/qualifier(s)] [= output-file]

where:

/qualifiers	Specify special actions the compiler is to perform.
file-spec-list	Specifies the source file(s) to be compiled. If you separate file specifications with a plus sign, BASIC compiles the files in the order you specify and creates a single object file from them. If you separate source file specifications with commas, BASIC compiles the programs separately and creates separate object files.
output-file	Specifies a name for the object file.

When you specify a file name argument to the DCL BASIC command, BASIC compiles the program and generates an object module with the specified file name and a file type of OBJ. The compiler can also generate other output files depending on the qualifiers you supply.

When you compile a source file with the DCL BASIC command and specify only its file name, the compiler searches for a source file with the specified name that:

1. Is stored on the default device
2. Is catalogued under the default directory name
3. Has a file type of BAS

If more than one file meets these conditions, the compiler chooses the one with the highest version number.

For example, assume that your default device is DBA0:, your default directory is SMITH, and your compiler command line is:

```
$ BASIC FIRSTTRY<RET>
```

The compiler searches device DBA0:, in directory [SMITH], seeking the highest version of FIRSTTRY.BAS. If you do not specify an output file, the compiler generates the file FIRSTTRY.OBJ and stores it on device DBA0: in directory [SMITH]; it then assigns the file a version number one higher than any other version.

1.4.2 Linking and Running Programs

The VAX–11 Linker uses the object module produced by BASIC as input and produces an executable image file as output. When your program is segmented—that is, when it is made up of more than one program module—the linker takes multiple object files and creates a single executable image from them. See Chapter 3 for more information on program segmentation.

You use the DCL LINK command to invoke the VAX–11 Linker. The LINK command has the format:

$ LINK[/command-qualifier(s)] file-spec-list[/file-qualifier(s)]

where:

command-qualifier(s)	Specifies output file options.
file-spec-list	Specifies a file or files to be linked.
file-qualifier(s)	Specifies input file options.

If you type LINK, the system prompts with:

```
$_File:
```

Respond by typing the file specification(s). If the file specifications do not fit on the line, type a hyphen (–) as the last character on the line and continue on the next line.

For example, to link the object file created from the FIRSTTRY program in Section 1.4.1, type:

```
$ LINK FIRSTTRY(RET)
```

This command tells the linker to accept FIRSTTRY.OBJ as input, and to produce FIRSTTRY.EXE as output. Once the executable file has been created, you run it with the DCL RUN command:

```
$ RUN FIRSTTRY(RET)

This program displays the product of two numbers.
Numbers? 5,6(RET)
Their product is 30
$
```

Chapter 2
Compiler Commands and Qualifiers

This chapter describes the use of compiler commands, in particular, the COMPILE command and its qualifiers, and the BASIC DCL command and its qualifiers. The qualifiers you use to compile a program in the BASIC environment differ slightly from those you use when compiling from DCL level. In addition, this chapter describes the default values for both COMPILE command qualifiers and the DCL BASIC command qualifiers.

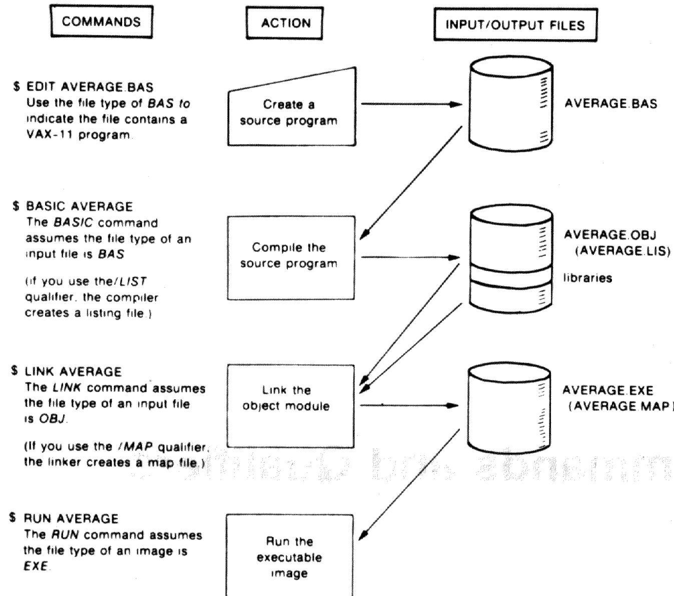
Most computer languages require you to perform the coding, compiling, linking, and executing steps separately; that is, you first create a program with a text editor, then compile it, then link it, then execute (or run) the program. If the program does not execute correctly, you must modify it and repeat these steps until it does.

BASIC lets you develop programs in this way, but in addition, you can develop a program in the BASIC environment. In this special environment, you can type in your program and execute it without first having to compile and link it. When your program is complete, you just type RUN, and BASIC automatically compiles, links, and executes the program.

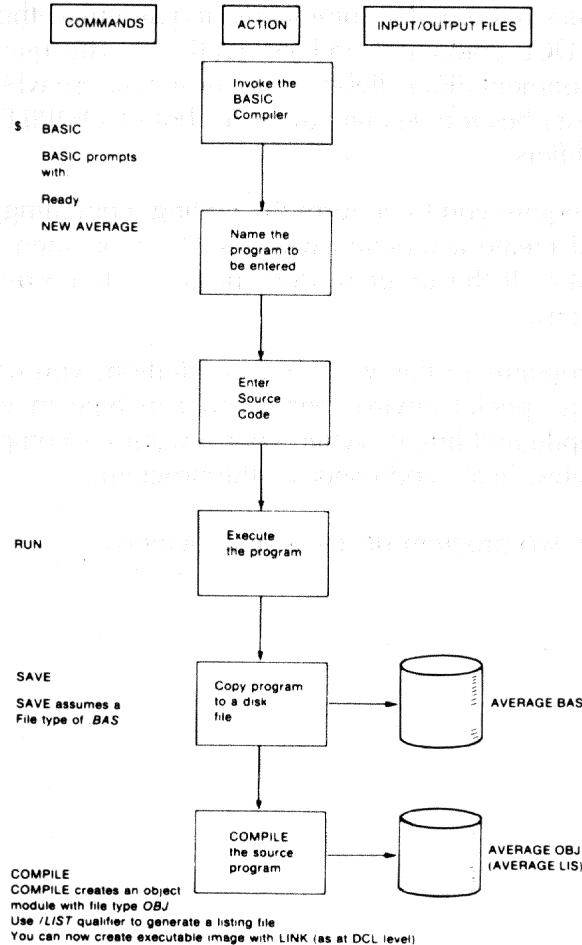
Figure 2–1 compares these two program development methods.

Figure 2–1: Program Development Methods

USING VAX-11 BASIC
AT DCL COMMAND LEVEL



IN THE BASIC ENVIRONMENT



MK–00896–00

The following sections describe the use of BASIC both from DCL command level and within the BASIC environment.

2.1 Using the BASIC Environment

When in the BASIC environment, you communicate directly with the BASIC compiler. You can enter:

- Line-numbered statements

  BASIC stores line-numbered statements as part of the current program in memory. If you enter a program line with the same number as an existing line, the new line replaces the old one. Note that line numbers are not required for every line. In fact, only the first line in a program requires a line number. See Chapter 1 in the *BASIC User's Guide* for more information about BASIC program lines.

- Compiler commands

  Compiler commands let you:

  - Display, edit, and merge BASIC source programs
  - Set compiler defaults
  - Move BASIC source programs to and from storage
  - Execute programs

- Immediate mode statements

  Immediate mode statements let you debug programs and perform calculations. See Chapter 1 in this manual for more information.

Many compiler commands accept file specifications as arguments. See Chapter 1 in this manual for the complete format of a file specification.

Compiler commands can be abbreviated to three letters and are not preceded by numbers or spaces. Table 2–1 shows a list of compiler commands. The following sections describe these commands in detail.

Table 2–1: BASIC Compiler Commands

Command	Function
APPEND	Merges the specified program with the program currently in memory.
ASSIGN	Assigns a logical name to complete file specification (the equivalence name).
$Command	Starts a subprocess to execute the specified DCL command.
COMPILE	Generates an object module from a BASIC source program. The object module has a default file type of OBJ.
CONTINUE	Resumes execution after a STOP statement or a CTRL/C.
DELETE	Erases a specified line(s) from a BASIC source program.
EDIT	Changes source text.
EXIT	Returns to DCL command level.
HELP	Displays HELP text.

(continued on next page)

Table 2–1: BASIC Compiler Commands (Cont.)

Command	Function
IDENTIFY	Causes BASIC to print an identification header on the terminal.
LIST	Displays the current source program on the terminal.
LISTNH	Displays the current source program without header information.
LOAD	Loads an object module into memory.
LOCK	Specifies default values for compiler command qualifiers (identical to the SET command).
NEW	Clears memory for the creation of a new program and assigns a new program name.
OLD	Reads a specified BASIC source program into memory.
RENAME	Changes the name of the program currently in memory.
REPLACE	Replaces a stored program with the program currently in memory. In VAX–11 BASIC, REPLACE is identical to SAVE.
RESEQUENCE	Supplies new line numbers for the program currently in memory.
RUN	Executes the program currently in memory, or a specified BASIC source program. The program in memory can be: 1) a BASIC source program placed in memory with the OLD command, 2) object module(s) placed in memory with the LOAD command, or 3) a combination of 1 and 2.
RUNNH	Identical to RUN but does not display header information.
SAVE	Creates a copy of the current source program on a specified device.
SCALE	Controls accumulated round-off errors for numeric operations.
SCRATCH	Erases the current program and any loaded object modules.
SEQUENCE	Generates line numbers for input text.
SET	Specifies default values for compiler command qualifiers.
SHOW	Displays the current default compiler qualifiers.
UNSAVE	Deletes a specified file.

The following sections describe these compiler commands.

2.1.1 APPEND

The APPEND command merges a BASIC source program with the program currently in memory. Its format is:

APPEND file-spec

where:

file-spec Is the source program to be appended.

The program in memory must be a BASIC source program, either 1) placed in memory with the OLD command or 2) entered in the BASIC environment. If both programs contain a line with the same number, the appended program line replaces the current program line.

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

```
Append file name--
```

Respond with a file name. If you respond by typing the RETURN key, BASIC searches for a file called NONAME with the default file type of BAS. If the compiler cannot find the file, it signals an error.

The APPEND command does not change the name of the program in memory.

2.1.2 ASSIGN

The ASSIGN command equates a logical name to a complete file specification, a device, or another logical name. Its format is:

ASSIGN equ-name[:] log-name[:]

where:

equ-name Is an equivalence name specifying the device or file specification to be assigned a logical name.

log-name Specifies a 1– through 63–character logical name to be associated with the file specification or device.

If the logical name translates to a device name and will be used in place of a device name in a file specification, terminate the equivalence name with a colon.

This example uses the ASSIGN command to make the system HELP library available from within BASIC:

```
Ready

ASSIGN SYS$HELP:HELPLIB HLP$LIBRARY
```

2.1.3 $ Command

You can enter a DCL command while in the BASIC environment by preceding it with a dollar sign ($). BASIC passes the command to the DCL for execution. The program currently in memory does not change.

Note that VAX–11 BASIC starts a subprocess to execute the command, and the command executes in the context of that subprocess. This can sometimes produce unexpected results. For example, a $ SET DEFAULT command typed in the BASIC environment sets the default for the subprocess but not for the process in which BASIC executes. The newly set default exists only until control returns to BASIC.

2.1.4 COMPILE

The COMPILE command creates an object module from a source program in memory. Its format is:

COMPILE[/qualifier] [file-name]

where:

file-name Specifies a name for the output file or files. Do not specify a file type; if you do, the listing file and object file will have the same file name and type. If you omit the file name, BASIC uses the current program name. The default file type for the object module is OBJ. The default file type for the listing file is LIS.

qualifier Specifies a COMPILE argument.

In the BASIC environment, qualifiers apply to the COMPILE command. For example, you can compile a program currently in memory and specify the creation of a listing file:

```
COMPILE/LIST
```

You can abbreviate all COMPILE qualifiers to four letters. See Table 2–2 for a list of qualifiers to the COMPILE command.

Table 2–2: COMPILE Command Qualifiers

Qualifier	Function
/[NO]ANSI_STANDARD	Tells BASIC to compile the program according to ANSI Minimal BASIC rules and to flag statements that do not conform to the ANSI Minimal BASIC standard.
/[NO]BOUNDS_CHECK	Tells BASIC to perform range checks on array subscripts. That is, it checks that all array references are to addresses within the array boundaries.
/BYTE	Specifies that integers not explicitly typed with a data type keyword use 8 bits of storage. This lets you use integer values between –128 and 127.
/[NO]CROSS[=[NO]KEYWORDS]	Causes BASIC to generate a cross-reference listing. If you specify KEYWORDS, BASIC provides a cross-reference list of BASIC keywords. The default is /NOCROSS. If you specify /CROSS, the default is /CROSS=NOKEYWORDS.
/[NO]DEBUG	Provides the debugger with local symbol definitions for program variables, constants, line numbers, and labels.
/DOUBLE	Specifies that floating point data use 64 bits of storage in D_float format. This lets you use floating-point values in the range $2.9 * 10^{-39}$ to $1.7 * 10^{38}$ and with up to 16 digits of precision.
/GFLOAT	Specifies that floating-point data use 64 bits of storage in G_float format. This lets you use floating-point values in the range $5.6 * 10^{-308}$ to $9.0 * 10^{309}$ and with up to 15 digits of precision.
/HFLOAT	Specifies that floating-point data use 128 bits of storage in H_float format. This lets you use floating-point values in the range $8.4 * 10^{-4933}$ to $5.9 * 10^{4933}$ and with up to 33 digits of precision.
/[NO]LINES	Enables the executing program to report the line number of statements causing errors and to use the RESUME statement without specifying a line number.

Table 2–2: COMPILE Command Qualifiers (Cont.)

Qualifier	Function
/[NO]LIST	Creates a program listing with a default file type of LIS.
/LONG	Specifies that untyped integers use 32 bits of storage. This lets you use integer values between –2147483648 and 2147483647.
/[NO]MACHINE	Includes the compiler-generated assembly code listing.
/[NO]OBJECT	Generates a linkable object module. This object module has the same file name as the BASIC source program and a default file type of OBJ.
/[NO]OVERFLOW { =[NO]INTEGER / =[NO]DECIMAL }	Enables the detection of arithmetic overflow for operations on integer or packed decimal data. If you do not supply a value, OVERFLOW affects both data types.
/[NO]ROUND	Specifies whether BASIC rounds or truncates packed decimal numbers.
/[NO]SETUP	/NOSETUP causes BASIC to optimize the executable image by omitting certain calls to the Run-Time Library at the start and end of each program unit. /SETUP is the default.
/SINGLE	Specifies that floating-point data use 32 bits of storage. This lets you use floating-point values in the range $2.9 * 10^{-39}$ to $1.7 * 10^{38}$ and with up to 6 digits of precision.
/[NO]SYNTAX_CHECK	Enables line-by-line syntax checking. Because BASIC automatically performs syntax checking when you compile a program, you normally use /SYNTAX_CHECK with the SET command to enable line-by-line syntax checking.
/[NO]TRACEBACK	Provides line numbers for the debugger and error reporter so they can translate virtual addresses into source program module names and line numbers.
/VARIANT=value	Provides a value to be tested in conditional compilations.
/[NO]WARNINGS= { [NO]WARNINGS / [NO]INFORMATIONALS }	Tells BASIC whether to display warning or informational error messages. /NOWARNINGS means that BASIC does not display any informational or warning errors.
/WORD	Specifies that all integer data not explicitly typed use 16 bits of storage. This lets you use integer values in the range –32768 to 32767.

2.1.5 CONTINUE

The CONTINUE command resumes program execution after BASIC encounters a STOP statement or a CTRL/C. Its format is:

CONTINUE

After a STOP statement or a CTRL/C is encountered in the BASIC environment, you can enter immediate mode statements to display or change program variables. Then, type CONTINUE to resume execution with the new values. See Chapter 1 for more information about immediate mode statements.

2.1.6 DELETE

The DELETE command removes a specified line or lines from the BASIC source program currently in memory. Its format is:

```
DELETE  { lin-num[,lin-num]. . . }
        { lin-num-lin-num        }
```

where:

lin-num   Is a program line number.

If you separate line numbers with commas, BASIC erases only those program lines. If you separate line numbers with a hyphen (–), BASIC erases those program lines and all program lines between them. For example:

DELETE 10	Removes line 10 from the program.
DELETE 50, 100	Removes lines 50 and 100 from the program.
DELETE 50, 100–190	Removes line 50 and lines 100 through 190 from the program.

If you do not specify a line number, DELETE has no effect.

2.1.7 EDIT

The EDIT command replaces text in the current BASIC program with text you supply in the command. If you type EDIT with no argument, VAX–11 BASIC invokes a text editor and reads the current program into the editor's buffer.

If you want to change text in the program without invoking an editor the EDIT format is:

```
EDIT [lin-num] [delim search-string delim replacement-string]
     [delim occurrence[,sub-line]]
```

where:

lin-num	Is the line you want to edit. If you do not specify a line number, the default is the last edited line. If there are no arguments to the EDIT command, EDIT invokes your default text editor.
delim	Can be any printing character not used in the search or replacement string.
search-string	Is the string you want to change or remove. If you do not specify a search string, BASIC displays the line but does not change it.
replacement-string	Is the string to be substituted for the search string. If you do not specify a replacement string, BASIC deletes the search string.
occurrence	Specifies which occurrence of search-string is replaced in the line. The default is the first occurrence.
sub-line	Is a number specifying a continuation of lin-num. Use sub-line for programs containing multiple lines of text with a single line number. BASIC skips (sub-line minus 1) lines before trying to match the search string.

For example:

EDIT 100 /LEFT$/RIGHT$/	Replaces the first occurrence of LEFT$ with RIGHT$ in line 100.
EDIT	Invokes the default editor and reads the current program into the editor's buffer.
EDIT 2000	Lists line 2000 (line 2000 becomes the default EDIT line).
EDIT 30 /LEFT$/RIGHT$/,3	Starts the search on the third text line of program line 30 and replaces the first occurrence of LEFT$ with RIGHT$.
EDIT 300/LEFT$(//2	Removes the second occurrence of LEFT$(from line 300. Note that you must specify delimiters around the null replacement string. Otherwise, the EDIT command would replace the first occurrence of LEFT$(with "2".

Typing EDIT with no argument causes BASIC to temporarily save your program in a file called:

```
BASEDITMP.TMP
```

BASIC then invokes the same editor you use when you type the DCL EDIT command. Exiting from the editor causes the changed program to become the new current BASIC program. BASIC then prompts with "Ready".

Note that BASIC deletes all versions of BASEDITMP.TMP when control returns from the editor.

2.1.8 EXIT

The EXIT command clears memory and returns control to DCL command level. Its format is:

EXIT

If you modify a program and issue the EXIT command before you copy it to disk with the SAVE or REPLACE command, BASIC signals "Unsaved change has been made, CTRL/Z or EXIT to exit". This message warns you that the edit will be lost if you do not save the program. You can then store the program or retype the EXIT command (or CTRL/Z) to exit from BASIC.

2.1.9 HELP

The HELP command lets you display the contents of the BASIC HELP library on the terminal. Its format is:

HELP [keyword] [keyword] [keyword] [keyword]

where:

keyword Is a BASIC keyword, command, or concept.

For example:

```
HELP STATEMENTS
```

This example causes BASIC to print a list of statements for which there is HELP available. BASIC then prompts with:

```
STATEMENTS Subtopic?
```

When you type the statement name in response to this prompt, BASIC prints: 1) an explanation of the statement's purpose, 2) the statement's general format, and 3) an example of its use.

When the keyword you enter is associated with other keywords, BASIC lists the additional keywords. For example:

```
STATEMENTS Subtopic? ON

ON

Additional information available:

ERROR      GOSUB      GOTO
```

You can then ask for additional information by typing:

```
STATEMENTS ON Subtopic?GOSUB
```

You can also display HELP text for BASIC errors. Typing HELP ERROR causes BASIC to display a list of 3– to 9–character error mnemonics.

For example, suppose your program invokes a user-defined DEF function with a null argument. This causes BASIC to signal "Actual argument must be specified". The actual error message looks like this:

```
%BASIC-E-ACTARGMUS, actual argument must be specified
```

You display the HELP text by typing:

```
HELP ERROR ACTARGMUS
```

BASIC displays:

```
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.
```

You can access run-time errors both with the mnemonic or the error number. You specify the error number with the letters "ERR" followed by the error number. For example, you can display the HELP text for the end-of-file error either by typing:

```
HELP ERROR ENDFILDEV
```

or

```
HELP ERROR ERR11
```

2.1.10 IDENTIFY

The IDENTIFY command prints a header containing the BASIC compiler name and version number. Its format is:

IDENTIFY

For example:

```
IDENTIFY

VAX-11 BASIC V2.0

Ready
```

2.1.11 LIST and LISTNH

The LIST and LISTNH commands display a specified line or lines. If you type LIST or LISTNH without specifying line numbers, BASIC displays a copy of the source program currently in memory, in ascending line number order. Its format is:

LIST[NH]

The LIST command causes BASIC to print a header displaying the program name and current time and date before displaying the specified lines. The LISTNH command suppresses the header information and prints the specified lines only. For example:

LIST 10	Displays header information, then displays line 10.
LISTNH 50, 100	Displays lines 50 and 100.
LIST 50, 90, 100–190	Displays header information, then displays lines 50, 90, and 100 through 190.

2.1.12 LOAD

The LOAD command makes an object module available for execution with the RUN command. Its format is:

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

where:

file-spec Is an object module generated by the BASIC compiler. The default file type is OBJ.

Note

You can load only object files created by BASIC.

If you do not specify a file-spec, the LOAD command erases any previously loaded object files.

The LOAD command accepts multiple device, directory, and file specifications. You separate multiple files with a plus sign. For example:

```
LOAD TEST1.OBJ+TEST2

Ready

RUN
```

The object files are not linked with the current program or executed until you issue the RUN command. Therefore, run-time errors in the loaded modules are not detected until you execute the program.

Each device and directory specification applies to all following file specifications until you specify a new directory or device. For example:

```
LOAD DB1:[SMITH]PROG3+[JONES]PROG4+DB2:PROG5
```

This command loads three object files: 1) PROG3 from Smith's directory on DB1:, 2) PROG4 from Jones' directory on DB1:, and 3) PROG5 from Jones' directory on DB2:.

2.1.13 LOCK

The LOCK command changes default values for COMPILE command qualifiers. It is equivalent to the SET command. Its format is:

LOCK /qualifier

where:

qualifier Is a COMPILE command qualifier.

For example:

```
LOCK /DOUBLE
Ready
```

This command specifies that all subsequent compilations use double-precision floating-point numbers. You can use any valid COMPILE command qualifier as an argument to LOCK.

2.1.14 NEW

The NEW command clears memory and assigns a name to a program to be entered. Its format is:

NEW program-name

where:

program-name Is the name of the program you want to create. You can specify only the program name. The name can be up to nine characters long.

For example:

```
NEW PROG1
```

This command assigns the name PROG1 to the program. You then enter program lines. If you do not specify a name, BASIC prompts with:

```
New file name--
```

Respond with a name. If you type only the RETURN key in response to the "New file name--" prompt, BASIC assigns the name NONAME.

2.1.15 OLD

The OLD command brings a previously created BASIC source file into memory. Its format is:

OLD file-spec

For example:

```
OLD PROG1
```

This command reads PROG1.BAS into memory.

If you do not specify a file name, BASIC prompts with:

```
Old file name--
```

Respond with a file name. If you do not specify a file type, BASIC reads a file with the specified file name and the default file type. The default file name is NONAME. Thus, if you type only the RETURN key in response to the "Old file name--" prompt, BASIC searches for NONAME.BAS.

2.1.16 RENAME

The RENAME command assigns a new name to the program currently in memory. Its format is:

RENAME program-name

where:

program-name Is the name of the program. You can specify only the program name.

For example, the following command sequence brings a program named PROG1 into memory and changes its name:

```
OLD PROG1

Ready

RENAME PROG2
```

The current program becomes PROG2. The disk file from which it was read is unchanged.

2.1.17 REPLACE

The REPLACE command writes the program in memory to a specified device. Its format is:

REPLACE [file-spec]

A REPLACE command always writes a copy of the current program to disk. If the file already exists, BASIC creates a new version.

The REPLACE command can also assign a new name to the edited program.

2.1.18 RESEQUENCE

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

RESEQUENCE [[lin-num1] [sep lin-num2] [int-const] [STEP num-exp]]

sep: { TO , ; }

where:

lin-num1	Is the line number in the current program where resequencing begins.
sep	Is a comma or semicolon.
int-const	Specifies the new first line number; the default number for the new first line is 100. If int-const will cause existing lines to be deleted or surrounded, BASIC signals an error.
num-exp	Specifies the numbering increment for the resequencing operation. The default for num-exp is 10.
lin-num2	Is the line number in the current program where the resequencing operation ends.

2.1.19 RUN and RUNNH

The RUN command executes a program. This program can be: 1) the current BASIC program, 2) object module(s) placed in memory with the LOAD command, 3) a combination of 1 and 2, or 4) a specific BASIC source program. Its format is:

RUN[NH] [file-spec]

If you do not supply a file specification, BASIC executes the program in memory:

```
Ready

OLD
Old file name--PROG1
Ready

RUN
```

The RUN command compiles, links, and executes PROG1. It prints a header displaying the program name and the current date and time. To execute a program without displaying this header, type RUNNH.

The RUN command does not create an object module file or a list file. It uses whatever qualifiers have been set, with the exception of those that make little sense when running in the BASIC environment. These are:

- NOCROSS
- NODEBUG
- NOLIST

- NOMACHINE
- NOOBJECT

These qualifiers are always in effect when you type RUN.

2.1.20 SAVE

The SAVE command copies a BASIC source program from memory to a disk file. Its format is:

SAVE [file-spec]

For example, if you type the following program, a SAVE command causes BASIC to arrange the program in ascending line number order, and copy it to a disk file on DB0:, in the current default directory, with file name TEST and the default file type of BAS:

```
30 PRINT 'THIS IS A TEST'
10 REM THIS IS A TEST
SAVE DB0:TEST

Ready
```

You can also specify a storage device, a file name, and a file type:

```
SAVE DB1:NEWTSTB.BBB
```

BASIC saves the program on disk DB1: in the current default directory, with a file name of NEWTST and a file type of BBB. If the program in memory has no name and you issue the SAVE command with no argument, BASIC copies the program to a file named NONAME with the default file type in your current default device and directory.

2.1.21 SCALE

The SCALE command can overcome accumulated round-off errors by multiplying double-precision floating-point values by 10 raised to the specified scale factor before storing them. Its format is:

SCALE int-const

where:

int-const Specifies the power of 10 you want to use as the scaling factor. It must be an integer value in the range from zero to six, inclusive.

2.1.22 SCRATCH

The SCRATCH command clears memory by: 1) resetting the program name to NONAME and 2) removing any object files previously loaded with the LOAD command. Its format is:

SCRATCH

2.1.23 SEQUENCE

The SEQUENCE command automatically generates line numbers for input text. Its format is:

SEQUENCE [starting-line-number] [,increment]

where:

starting-line-number — Is the first line number to be entered. If you do not specify a starting line number, the default is the last line inserted by a SEQUENCE command. If there is no previous SEQUENCE command, the default is line 100.

increment — Is the number added to each line number to get the new line number. The default is 10.

After a SEQUENCE command, BASIC prompts with a line number and prompts again after each source line you enter. If you enter a CTRL/Z (either in response to the line number prompt or at the end of a program line), BASIC stops prompting and you can enter source text in the normal way. If you specify a starting line number that already contains a statement, BASIC signals "Attempt to sequence over existing statement" and returns to normal input mode.

2.1.24 SET

The SET command specifies defaults for compiler command qualifiers. Its format is:

SET /qualifier

where:

qualifier — Is a COMPILE command qualifier.

For example:

```
SET /SINGLE

Ready
```

This command makes /SINGLE the default for the COMPILE or RUN command. Note that typing SET with no argument resets the defaults to their state at the time of entry into the BASIC environment.

2.1.25 SHOW

The SHOW command displays the current default qualifiers and user libraries. Its format is:

SHOW

For example:

```
SHOW

VAX-11 BASIC V2.0 Current Environment Status 22-JUN-1982 10:12:12.05
DEFAULT DATA TYPE INFORMATION:                LISTING FILE INFORMATION INCLUDES:
    Data type : REAL                              NO Source
    Real size : SINGLE                            NO Cross reference
    Integer size : LONG                              CDD Definitions
    Decimal size : (15,2)                            Environment
    Scale factor : 0                              NO Override of %NOLIST
    NO Round decimal numbers                      NO Machine code
                                                     Map
                                                     INCLUDE files
```

```
COMPILATION QUALIFIERS IN EFFECT:
        Object file
        Overflow check integers              FLAGGERS:
     NO Overflow check decimal numbers              Declining features
        Bounds checking                          NO BASIC PLUS 2 subset
     NO Syntax checking
        Lines
        Variant : 0                          DEBUG INFORMATION:
     NO Warnings                                    Traceback records
     NO Informationals                           NO Debug symbol records
        SETUP
        Object Libraries : NONE
Ready
```

This information tells you what will be produced by a COMPILE command.

The DEFAULT DATA TYPE INFORMATION display tells you that:

- The default data type is REAL.
- The default size for floating-point numbers is SINGLE, the default size for integers is LONG, and the default size for packed decimal numbers is (15,2).
- There is no scale factor in effect.
- Packed decimal numbers are truncated rather than rounded.

The LISTING FILE INFORMATION display tells you which parts of the program listing will be included if you create a compilation listing. This means:

- The source program will not be listed.
- No cross-reference information will be listed.
- CDD definitions will be displayed as BASIC RECORD statements.
- The qualifiers in effect when the program was compiled will be listed. This means that the program listing will contain the equivalent of this SHOW command.
- The %NOLIST compiler directive will not be overridden.
- No compiler-generated machine code will be listed.
- An allocation map will be listed. This contains the sizes and offsets of any variables.
- Files accessed with the %INCLUDE directive will be listed.

The COMPILATION QUALIFIERS IN EFFECT section tells you that:

- An object file will be produced.
- Overflow checking for integers is enabled.
- Overflow checking for packed decimal numbers is disabled.
- Bounds checking is enabled.
- Line-by-line syntax checking is disabled.

- Line number information will be included in the object file.
- The VARIANT value is zero.
- BASIC performs normal initialization calls at run time (SETUP).
- No warning or informational error messages will be displayed.
- No user-supplied object module libraries will be searched.

The FLAGGERS section tells you:

- Declining features will be reported.
- BASIC will not warn you if you use language elements that are not supported in PDP–11 BASIC–PLUS–2.

The DEBUG INFORMATION section tells you:

- Traceback information is included in the object module.
- No debug records are included in the object module. This means you cannot access program symbols with the VAX–11 Symbolic Debugger.

See Chapter 10 for more information about user libraries.

2.1.26 UNSAVE

The UNSAVE command deletes the specified version of a file from disk. Its format is:

```
UNSAVE file-spec
```

If you do not specify a file, UNSAVE deletes the disk file associated with the program currently in memory. If you do not specify a version number, UNSAVE deletes the newest version. For example:

```
OLD PROG1

Ready

UNSAVE

Ready
```

The OLD command copies a program named PROG1.BAS from disk to memory. The UNSAVE command deletes the program from disk.

You can delete a BASIC source program other than the one in memory by specifying the program name:

```
UNSAVE PROG2
```

This command deletes the most recent version of the file PROG2.BAS.

To delete a file other than a source program, specify the file name and file type. For example:

```
UNSAVE PROG2.OBJ
```

This command deletes the newest version of the object module generated from the compilation of PROG2.

2.2 Using BASIC from DCL Command Level

When developing BASIC programs from DCL command level, you:

- Create a BASIC source program with a text editor
- Generate an object module from the source file with the $ BASIC command
- Generate an executable image from the object module using the $ LINK command
- Run the executable image using the $ RUN command

Note

If you use the BASIC compiler in this way, you cannot use immediate mode statements to debug the program.

When using BASIC from DCL command level, you can specify compiler defaults with qualifiers to the BASIC command. Qualifiers have the form:

/qualifier[=value]

Many qualifiers have a corresponding form that negates the action specified by the qualifier. The negative form is:

/NOqualifier

For example, /LIST tells the compiler to produce a listing file; /NOLIST suppresses the listing.

You can specify qualifiers so that they affect either: 1) all files in the command or 2) only certain files. If the qualifier immediately follows the command name, it applies to all files:

```
$ BASIC/LIST ABC,XYZ,RST
```

This command specifies listing files for ABC, XYZ, and RST.

Qualifiers following a file specification (with some exceptions) affect only the associated file:

```
$ BASIC/LIST ABC,XYZ/NOLIST,RST
```

This command specifies listing files for ABC and RST, but not for XYZ. Qualifiers to one file specification in an appended list of file specifications are exceptions to this rule. See Example 5 in the following list.

Examples:

1. `$ BASIC/LIST AAA,BBB,CCC`

   BASIC compiles source files AAA.BAS, BBB.BAS, and CCC.BAS as separate files and produces three object files (AAA.OBJ, BBB.OBJ, and CCC.OBJ) and listing files (AAA.LIS, BBB.LIS, and CCC.LIS).

2. `$ BASIC XXX+YYY+ZZZ`

   BASIC appends source files XXX.BAS, YYY.BAS, and ZZZ.BAS and compiles them as a single program. This command produces one object file named XXX.OBJ.

3. `$ BASIC/OBJECT=SQUARE CIRCLE`

   BASIC compiles source file CIRCLE.BAS and produces object file SQUARE.OBJ. This command produces no listing file.

4. `$ BASIC AAA+BBB,CCC/LIST`

   BASIC produces two object files: 1) AAA.OBJ (created from AAA.BAS and BBB.BAS) and 2) CCC.OBJ (created from CCC.BAS). BASIC also produces the listing file CCC.LIS.

5. `$ BASIC ABC+CIRC/NOOBJECT+XYZ`

   BASIC appends and compiles the source files ABC.BAS, CIRC.BAS, and XYZ.BAS. Because qualifiers in a list of appended files affect all files in the list, this command suppresses the creation of an object file.

The following sections describe the qualifiers to the $ BASIC command. Table 2–3 lists the qualifiers you can use with the BASIC command. The following sections describe each qualifier in detail.

Table 2–3: Qualifiers of the DCL BASIC Command

Qualifier	Negative Form	Default
/ANSI_STANDARD	/NOANSI_STANDARD	/NOANSI_STANDARD
/CHECK = { [NO]BOUNDS, [NO]OVERFLOW, ALL, NONE }	/NOCHECK	/CHECK = (BOUNDS,OVERFLOW)
/CROSS	/NOCROSS	/NOCROSS
/DEBUG = { [NO]SYMBOLS, [NO]TRACEBACK, ALL, NONE }	/NODEBUG	/DEBUG = (TRACEBACK,NOSYMBOLS)
/DECIMAL_SIZE = (d,s)	None	/DECIMAL_SIZE = (15,2)
/DOUBLE *		
/FLAG = { [NO]BP2COMPATIBILITY, [NO]DECLINING }	/NOFLAG	/FLAG = DECLINING
/INTEGER_SIZE = { BYTE, WORD, LONG }	None	/INTEGER_SIZE = LONG
/LINES	/NOLINES	/LINES
/LIST[= file-spec]	/NOLIST	/NOLIST (from terminal) /LIST (batch)
/LONG *		
/MACHINE	/NOMACHINE	/NOMACHINE
/OBJECT[= file-spec]	/NOOBJECT	/OBJECT
/REAL_SIZE = { SINGLE, DOUBLE, GFLOAT, HFLOAT }	None	/REAL_SIZE = SINGLE

Table 2-3: Qualifiers of the DCL BASIC Command (Cont.)

Qualifier	Negative Form	Default
/SCALE=n	None	/SCALE=0
/SHOW= { CDD_DEFINITIONS, ENVIRONMENT, INCLUDE, MAP, OVERRIDE, SOURCE, MACHINE_CODE }	/SHOW= { NOCDD_DEFINITIONS, NOENVIRONMENT, NOINCLUDE, NOMAP, NOOVERRIDE, NOSOURCE, NOMACHINE_CODE }	CDD_DEFINITIONS ENVIRONMENT INCLUDE MAP NOOVERRIDE SOURCE NOMACHINE_CODE
/SINGLE *		
/SYNTAX_CHECK	/NOSYNTAX_CHECK	/NOSYNTAX_CHECK
/TYPE_DEFAULT= { INTEGER, REAL, DECIMAL, EXPLICIT }	None	/TYPE_DEFAULT=REAL
/VARIANT=value	None	/VARIANT=0
/WORD *		

* The SINGLE, DOUBLE, WORD, and LONG qualifiers are supported for compatibility with older versions of VAX-11 BASIC. However, DIGITAL recommends that you use the /TYPE_DEFAULT, /INTEGER_SIZE, /REAL_SIZE, and /DECIMAL_SIZE qualifiers to set the default data type and size.

Note that you can also use these qualifiers to set defaults when entering the BASIC environment. For example:

```
$ BASIC/TYPE_DEFAULT=INTEGER/REAL_SIZE=DOUBLE/SYNTAX_CHECK

VAX-11 BASIC V2.0

Ready
```

This command causes you to enter the BASIC environment with the default data type set to INTEGER, the default size for floating-point numbers set to DOUBLE, and with automatic line-by-line syntax checking enabled.

The following sections describe qualifiers to the DCL BASIC command in more detail.

2.2.1 ANSI_STANDARD

ANSI_STANDARD causes BASIC to allow only statements valid for ANSI Minimal BASIC and to compile programs according to the ANSI Minimal BASIC rules. Its format is:

/ANSI_STANDARD

See Chapter 7 for more information about ANSI standard BASIC.

2.2.2 AUDIT

AUDIT tells BASIC to include a history entry in the CDD when extracting a CDD definition. Its format is:

/[NO]AUDIT= { str-lit, file-spec }

where:

str-lit	Is a quoted string.
file-spec	Is a file specification.

If you specify a quoted string, 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 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 the CDD

2.2.3 CHECK

CHECK causes BASIC to test for arithmetic overflow and for array references outside array boundaries when the program executes. The CHECK qualifier format is:

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

where:

BOUNDS	Checks array subscripts to ensure that they are within array boundaries specified by the program.
OVERFLOW	Enables the detection of arithmetic overflow for integer and packed decimal operations.
ALL	Specifies that both OVERFLOW and BOUNDS checking are performed.
NONE	Specifies that neither OVERFLOW or BOUNDS checking is performed.

The qualifier /CHECK is equivalent to /CHECK=ALL, and /NOCHECK is equivalent to /CHECK=NONE. If you specify /CHECK=OVERFLOW, overflow checking is enabled for both integers and packed decimal numbers. Similarly, specifying /CHECK=NOOVERFLOW disables overflow checking for both types of numbers.

Specifying /NOBOUNDS means that your program is smaller and runs faster. However, no error is signaled for an array reference outside the bounds of an array. This means that the program may get a memory management or access violation error at run time. Therefore, this option should be used only for programs that have been thoroughly debugged and whose execution time is critical.

2.2.4 CROSS

/CROSS tells the compiler to generate a cross-reference listing. Its format is:

/CROSS[=KEYWORDS]

The cross-reference list shows program symbols, their class, and the program lines in which they are referenced. /NOCROSS specifies that no cross-reference listing is produced. See Chapter 11 in the *BASIC User's Guide* for more information on cross-reference listings.

/CROSS=KEYWORDS specifies that the cross-reference listing includes all references to BASIC keywords. If you specify /CROSS, the default is NOKEYWORDS.

2.2.5 DEBUG

/DEBUG tells the compiler to provide information for the VAX–11 Symbolic Debugger and the system run-time error traceback mechanism. Its format is:

```
          [NO]SYMBOLS
/DEBUG= { [NO]TRACEBACK }
          ALL
          NONE
```

where:

SYMBOLS	Tells the compiler to provide the debugger with local symbol definitions for user-defined variables (including dimension information for arrays).
TRACEBACK	Tells the compiler to provide an address correlation table so the debugger and the run-time error traceback mechanism can translate absolute addresses into source program routine names and line numbers.
ALL	Specifies that the compiler provides both local symbol definitions and an address correlation table.
NONE	Specifies that the compiler provides no debugging information.

The default is /DEBUG=TRACEBACK. The qualifier /DEBUG is equivalent to /DEBUG=ALL, and /NODEBUG is equivalent to /DEBUG=NONE. For more information on debugging, see Chapter 4.

Neither TRACEBACK nor SYMBOLS affects a program's executable code.

2.2.6 DECIMAL_SIZE

DECIMAL_SIZE lets you specify the default size for packed decimal data. Its format is:

```
/DECIMAL_SIZE=(d,s)
```

where:

d Is the total number of digits in the number.

s Is the number of digits to the right of the decimal point.

The 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. The default packed decimal size is (15,2). See Chapter 5 in the *BASIC User's Guide* for more information about packed decimal numbers.

2.2.7 FLAG

FLAG lets you specify whether BASIC warns you about declining features and compatibility with PDP–11 BASIC–PLUS–2. Its format is:

```
                   { [NO]BP2COMPATIBILITY }
/ [NO]FLAG =       { [NO]DECLINING        }
                   { ALL                  }
                   { NONE                 }
```

/FLAG = BP2COMPATIBILITY causes BASIC to warn you if you use a language element that is not supported in PDP–11 BASIC–PLUS–2. /FLAG = DECLINING causes BASIC to warn you if you use a language feature that is not recommended.

/FLAG is equivalent to /FLAG = ALL and /NOFLAG is equivalent to /FLAG = NONE.

2.2.8 INTEGER_SIZE

INTEGER_SIZE lets you specify the default size for integer data. Its format is:

```
                      { LONG }
/INTEGER_SIZE =       { WORD }
                      { BYTE }
```

The default integer size applies to all integer variables whose data type is not explicitly declared. The default is LONG. See Chapter 5 in the *BASIC User's Guide* for more information about integer data types.

2.2.9 LINES

LINES makes line number information available for the ERL function, the RESUME statement (with no line number), and the BASIC error reporter. Its format is:

```
/LINES
```

/NOLINES disables this function. If your program contains a RESUME statement with no line number argument, or a reference to the error-handling function ERL, the compiler overrides NOLINES and signals "ERL overrides /NOLINE".

Note

The BASIC error reporting facility is separate from that of system traceback.

Specifying /NOLINES makes your program run faster and reduces program size (this eliminates five bytes of code and four bytes of data for each program line number). However, if you specify /NOLINES, you: 1) cannot use the ERL function, and 2) do not receive the line number in error messages. Therefore, this option should be used only for programs that have been thoroughly de-bugged and whose execution time is critical.

2.2.10 LIST

LIST tells BASIC to produce a source listing file. Its format is:

```
/LIST[ = file-spec]
```

You can include a file specification for the listing file. Otherwise, the output file defaults to the name of the first source file and a file type of LIS.

At a terminal, the default is /NOLIST. In batch mode, the default is /LIST.

Note that the /LIST qualifier only controls whether or not BASIC produces a listing file. The /SHOW qualifier controls which parts of the listing are produced.

2.2.11 MACHINE

MACHINE specifies that the listing file includes the compiler-generated object code. Its format is:

```
/MACHINE
```

This qualifier is ignored if no listing file is generated. /NOMACHINE is the default.

2.2.12 OBJECT

OBJECT tells BASIC to produce an object module, and optionally specifies its file name. Its format is:

```
/OBJECT[=file-spec]
```

The default is /OBJECT.

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 (see examples in Section 2.1).

To produce an object file with an explicit file specification, you must use the /OBJECT qualifier, in the form /OBJECT=file-spec. Otherwise, the object file has the same name as its corresponding source file, and a file type of OBJ. By default, the object file produced from appended source files has the name of the first source file specified. All other file specification attributes (node, device, directory, and version) assume the default values.

During the early stages of program development, you may find it helpful to suppress the production of object files until your source program compiles without errors. Use the /NOOBJECT qualifier to do this.

2.2.13 REAL_SIZE

REAL_SIZE lets you specify the default size for floating-point data. Its format is:

```
               { SINGLE }
/REAL_SIZE=    { DOUBLE }
               { GFLOAT }
               { HFLOAT }
```

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

See Chapter 5 in the *BASIC User's Guide* for more information on floating-point data types.

2.2.14 ROUND

ROUND specifies that BASIC is to round packed decimal numbers rather than truncating them. Its format is:

```
/[NO]ROUND
```

2.2.15 SCALE

SCALE specifies a scale factor between zero and six, inclusive. The scale factor affects only double-precision numbers. The SCALE format is:

```
/SCALE=n
```

SCALE helps to control accumulated round-off errors by multiplying floating-point values by 10 raised to the scale factor before storing them in variables. /SCALE is ignored for all but double-precision floating-point numbers.

Note

> SCALE is provided for compatibility with existing programs and with other implementations of BASIC. DIGITAL recommends that you do not use this feature for new program development. Accumulated round-off errors can be better controlled with packed decimal numbers. See Chapter 5 in the *BASIC User's Guide* for more information on packed decimal numbers.

2.2.16 SHOW

SHOW determines which parts of the compilation listing are created. The /LIST qualifier must be in effect for SHOW to have any effect. Its format is:

```
/SHOW=(show-clause,...)
```

where show-clause can be any of the following:

```
[NO]CDD_DEFINITIONS

[NO]ENVIRONMENT

[NO]INCLUDE

[NO]MAP

[NO]OVERRIDE
```

The CDD_DEFINITIONS clause controls whether the translation of a CDD record is displayed in the listing.

The ENVIRONMENT clause lets you display all defaults that were in effect when the program was compiled. This is the compilation listing equivalent of the SHOW compiler command.

The INCLUDE clause controls whether files accessed with the %INCLUDE directive are displayed in the listing.

The MAP clause determines whether the listing contains an allocation map. The allocation map lists all program variables, their size and their data type.

The OVERRIDE clause helps you debug code by disabling the effect of the %NOLIST directive.

2.2.17 SYNTAX_CHECK

SYNTAX_CHECK tells BASIC to perform line-by-line syntax checking. Its format is:

```
/SYNTAX_CHECK
```

When syntax checking is enabled, BASIC immediately checks the syntax of every text line as soon as you type a carriage return. When syntax checking is disabled, BASIC does not perform syntax checking until you COMPILE or RUN the program.

2.2.18 TYPE_DEFAULT

TYPE_DEFAULT lets you specify the default data type for numeric variables. Its format is:

```
                    { INTEGER  }
/TYPE_DEFAULT =     { REAL     }
                    { DECIMAL  }
                    { EXPLICIT }
```

Specifying EXPLICIT means that all program variables must be explicitly declared in DECLARE, EXTERNAL, COMMON, MAP or DIM statements. Specifying INTEGER, REAL, or DECIMAL means only that variables and data are integer, real or packed decimal. To specify the actual size of variables and data, use the INTEGER_SIZE, REAL_SIZE and DECIMAL_SIZE qualifiers.

2.2.19 VARIANT

VARIANT lets you specify the value associated with the lexical function %VARIANT. Its format is:

```
/VARIANT=value
```

where:

value Is a longword integer value.

See Chapter 11 in the *BASIC User's Guide* for more information about VARIANT and the %VARIANT lexical function.

2.2.20 WARNING

WARNING lets you specify whether BASIC will display informational and warning error messages. Its format is:

```
/[NO]WARNINGS =   { [NO]WARNINGS        }
                  { [NO]INFORMATIONALS  }
```

Specifying /NOWARNINGS causes BASIC not to display any informational or warning errors. Specifying /WARNINGS=NOWARNINGS causes BASIC to display informational errors but not warning errors. Specifying /WARNINGS=NOINFORMATIONALS causes BASIC to display warning errors but not informationals. By default, BASIC displays both warning and informational errors.

2.2.21 Default Compiler Options

When you compile a program, you can choose options like /LIST (specifying a program listing file), or /DOUBLE (specifying double-precision floating-point numbers). The options you get when you don't specify them are called defaults.

You can change these defaults for your own programs with: 1) qualifiers to the BASIC DCL command, 2) qualifiers to the BASIC COMPILE command, and 3) the SET command in the BASIC environment.

To display the current defaults, you can enter the BASIC environment and use the SHOW command, as described in Section 2.1.24. Similarly, when compiling a program from DCL level, you can use the /SHOW=ENVIRONMENT qualifier to display current defaults.

The BASIC DCL command accepts qualifiers to either: 1) change the defaults for a single compilation or 2) change the defaults when entering the BASIC environment. For example:

```
$ BASIC/DOUBLE/LIST/NOOBJECT MYPROG
$ BASIC/DOUBLE/LIST/NOOBJECT
Ready
```

The first BASIC DCL command invokes BASIC to compile a single source file (MYPROG.BAS), and overrides the default compiler settings for:

1. Floating-point numbers, specifying double precision
2. Listing, specifying a compilation list
3. Object code output, specifying that an OBJ file not be produced

The second BASIC DCL command places you in the BASIC environment with the following defaults set:

1. Double-precision floating-point numbers
2. Compilation listing produced by the COMPILE command
3. No OBJ file produced by the COMPILE command

When in the BASIC environment, you can override defaults with either the SET command or a qualifier to the COMPILE command. The SET command locks the default to the specified value; the new default remains in effect until you either exit from BASIC or enter a new SET command for that default. When you set a default with a COMPILE command qualifier, the new default remains in effect only for that compilation. For example:

```
Ready
COMPILE/DOUBLE/LIST/NOOBJECT
```

or

```
Ready
SET /DOUBLE
Ready

SET /LIST
Ready

SET /NOOBJECT
Ready

SET /OBJECT/NOLIST
```

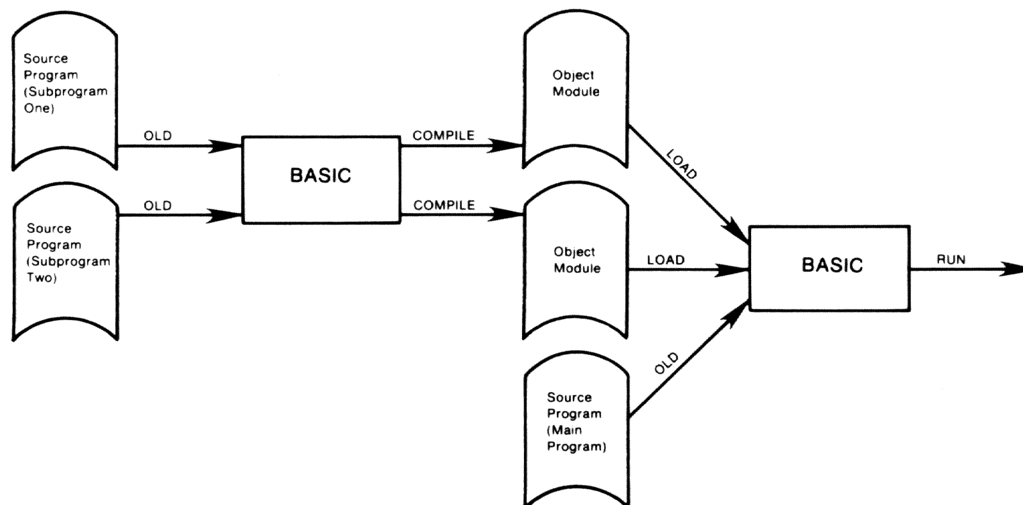
2.3 Running and Debugging Multiple Program Units

BASIC lets you execute and debug subprograms while in the BASIC environment. You do this in four steps:

1. Compile all subprograms, generating object modules.
2. Use the OLD command to read the main program into memory.
3. Use the LOAD command to bring the subprogram object modules into memory.
4. Type RUN.

See Figure 2–2.

Figure 2–2: Running Multiple Program Units



To execute this multi-module program in the BASIC environment, type:

```
OLD SB1
Ready
COMPILE
Ready
OLD MAIN
Ready
LOAD SB1
Ready
RUN
NOW IN MAIN PROGRAM
NOW IN SUBPROGRAM
BACK IN MAIN PROGRAM
Ready
```

For example, here are two simple BASIC programs:

Main Program: MAIN.BAS

```
10 REM THIS PROGRAM CALLS SUBPROGRAM SB1
20 PRINT 'NOW IN MAIN PROGRAM'
30 CALL SB1
40 PRINT 'BACK IN MAIN PROGRAM'
50 END
```

Subprogram: SB1.BAS

```
10 SUB SB1
20 PRINT 'NOW IN SUBPROGRAM'
30 SUBEND
```

When you run multiple program units in the BASIC environment, only one module is "currently compiled." Normally, the currently compiled program is the one you read into memory with the OLD command. However, if a source file contains more than one program module, the last one (that is, the one closest to the end of the source file) is the currently compiled module. In the previous example, MAIN is the currently compiled module.

If a STOP statement or CTRL/C is encountered in a module other than the currently compiled module, BASIC signals "Compiled procedure is currently not active". At this point, you cannot use immediate mode statements.

Chapter 3
Program Segmentation

Program segmentation is the process of dividing a program into small, manageable routines and modules. Each routine or module should contain only the code to perform one logical function. Coding programs in this way means:

- Faster program design and implementation
- Faster debugging and testing
- Easily understood code
- Easily maintained code
- Easily transported code

This chapter describes:

- How to declare subprograms using the EXTERNAL statement
- How to write subprograms using the SUB, END SUB, and EXIT SUB and FUNCTION, EXIT FUNCTION and END FUNCTION statements
- How to share data among program units using COMMON and MAP statements and subprogram parameters

You can also divide your program into more manageable parts using DEF function definitions and GOSUB blocks. See Chapter 6 in the *BASIC User's Guide* for an explanation of DEF function definitions, and Chapter 3 in the *BASIC User's Guide* for a description of GOSUB blocks.

Any subprogram processed by the VAX–11 BASIC compiler conforms to the VAX–11 Procedure Calling Standard. This standard prescribes how arguments are passed, how values are returned, and how procedures receive and return control. A BASIC subprogram can call or be called by any procedure written in a language that conforms to this standard.

3.1 Program Segmentation Techniques

A program module is a block of code that can be compiled as a unit. BASIC program modules include main programs and SUB and FUNCTION subprograms. A *routine* in BASIC is a block of code that is not compiled separately; it is part of a program module. Some examples of BASIC routines are DEF function definitions and GOSUB blocks.

Program modules and routines let you divide a program into small, manageable parts based on the logical function of those parts. For example, CALL and GOSUB statements both transfer control to another part of an executable image. After processing is complete, END SUB and RETURN statements both return control to the statement following the statement that transferred control. Similarly, DEF and FUNCTION statements both let you define functions that return values when invoked.

The difference between these segmentation techniques lies in the way the parts of the program interact. When you divide a large program into small, separately compiled program modules, the interface between these modules is strictly defined and well controlled. Variables and data can be shared between program modules only by way of formal parameters, COMMON or MAP blocks, or files. Thus, it is much more difficult for one program module to cause unexpected side effects in another program module.

The interface between a DEF or GOSUB block and the surrounding program is much less clearly defined; all program variables and data are accessible from within a DEF or GOSUB block. Thus, it is much easier for a DEF or GOSUB to cause unwanted side effects.

This is not to say that you should never use DEFs or GOSUBs. However, you should be aware of the possibility of side effects in these blocks and be more careful in coding them. Considering possible side effects is even more important if a program may be modified by other programmers.

These kinds of operations should be considered for segmentation into program modules:

- References to VAX/VMS System Services
- References to VAX–11 Run-Time Library routines
- Commonly used low-level routines, for example:
  - Date conversion procedures
  - Table look-ups
  - Rounding functions
  - Terminal I/O procedures
  - Menu processing procedures
- Programs too big to edit conveniently
- Program modules that are often modified
- File I/O procedures, such as OPEN

VAX/VMS System Services and VAX–11 Run-Time Library routines are specific to VAX/VMS systems; therefore, you should consider isolating references to them in subprograms. This makes it easier

to move the program to a different system. You should consider isolating commonly used routines because: 1) they can be shared by many programs and 2) they may also be specific to a particular operating system. You should consider isolating file I/O functions in subprograms so that changes to the system's hardware configuration cause changes only in a few program modules.

3.2 Declaring Subprograms

You declare the existence of a subprogram by naming it in an EXTERNAL statement in the calling program. Declaring subprograms is optional for SUB subprograms; the CALL statement that invokes a SUB subprogram is an implicit declaration. However, function subprograms must be declared with the EXTERNAL statement.

The EXTERNAL statement lets you declare the data type and parameter passing mechanism for each parameter passed to the subprogram. BASIC will convert actual arguments to the data type specified in the EXTERNAL statement, if necessary. If the subprogram is a function, the EXTERNAL statement also lets you specify the data type of the returned value. For subprogram declarations, the format of the EXTERNAL statement is:

```
EXTERNAL [data-type] { FUNCTION }  {sub-nam [([external-param],... )]},...
                     { SUB      }

external-param: [data-type] [DIM([,]...)] [=int-const] [pass-mech]

           { BY DESC  }
pass-mech: { BY REF   }
           { BY VALUE }
```

where:

data-type	Is a valid VAX–11 BASIC data type keyword or the name of a RECORD definition. You can specify data-type only for FUNCTION subprograms.
SUB	Specifies that the external procedure is a subprogram conforming to the VAX–11 Procedure Calling Standard. External procedures declared as SUB subprograms must be invoked with the CALL statement.
FUNCTION	Specifies that the external procedure is a subprogram which conforms to the VAX–11 Procedure Calling Standard and which returns a value. External procedures declared as FUNCTION subprograms can be invoked either as a function or with the CALL statement. However, if you invoke it with the CALL statement, no value is returned.
sub-nam	Is the name of the external procedure.
DIM([,]...)	Specifies the number of dimensions of an array passed as a parameter. The number of dimensions in the array is one more than the number of commas appearing inside the parentheses.
BY VALUE BY REF BY DESC	Is a parameter passing mechanism. See Section 3.3.2 for a discussion of parameter passing mechanisms. You should use these clauses only when calling programs written in a language other than BASIC.

If a BY clause appears before the parameter list, that parameter passing mechanism applies to all parameters in the list. However, this can be overridden by another BY clause after an individual parameter.

Data type keywords in the parameter list apply only to a single parameter. If you do not specify a parameter's data type, the default is determined by either: 1) a BASIC compilation qualifier (for example, /REAL_SIZE=DOUBLE) or 2) an OPTION statement. The OPTION statement overrides any compilation qualifiers. See Chapter 5 in the *BASIC User's Guide* for more information on the OPTION statement.

By declaring all subprograms before calling them, you gain several advantages:

- The program is better documented.
- Data types and passing mechanisms need to be specified only once.
- BASIC automatically converts parameters to the proper data type before passing them.
- When calling programs written in other languages, BASIC automatically uses the mechanism specified in the declaration for all calls to the subprogram.

The practice of declaring subprograms and specifying the data-type of parameters becomes more important when calling programs written in other languages. This is because other languages may not use the BASIC default parameter passing mechanisms. Thus it is much easier to declare data types and passing mechanisms only once, and then let BASIC automatically supply them for subsequent calls to the subprogram.

Again, you should not specify parameter passing mechanisms when calling a BASIC subprogram from another BASIC program; the BY clauses are for use only when calling non–BASIC subprograms.

These are sample subprogram declarations using the EXTERNAL statement:

```
100     EXTERNAL SUB PASCAL_SUB BY REF (LONG , LONG)
200     EXTERNAL GFLOAT FUNCTION MY_SUB (GFLOAT, LONG, GFLOAT)
300     EXTERNAL REAL FUNCTION DETERMINANT (LONG DIM(,))
```

Note that the parameter lists contain only data type and dimension information. They cannot contain any formal or actual parameters. When the external procedure is invoked, BASIC makes sure that the actual parameter data type and passing mechanism matches those specified in the EXTERNAL declaration. However, you must ensure that the parameters declared in the EXTERNAL statement match those in the external routine.

3.3 Calling Subprograms

The following sections describe how to:

- Invoke subprograms
- Pass parameters to subprograms
- Share data among program modules

3.3.1 Invoking Subprograms (CALL Statement)

The CALL statement transfers control to a subprogram, and optionally passes arguments to it. Its format is:

```
                  { BY VALUE }              { BY VALUE }
CALL sub-nam      { BY REF   }  [(param     { BY REF   }   [, ...])]
                  { BY DESC  }              { BY DESC  }
```

The subprogram name must be a unique, 1– to 31–character name.

The parameters in the CALL statement specify variables, constants, expressions, array elements, or arrays to be passed to the subprogram. You can also specify a function in the argument list; when you do this, BASIC passes the value returned by the function to the subprogram.

Here is a simple example of a BASIC main program calling a BASIC subprogram:

Main Program

```
100     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
```

Subprogram

```
100     SUB SUB01 (LONG X, LONG Y, LONG Z)
        PRINT 'The sum is'; X + Y + Z
        END SUB
```

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.

This example performs the same task using a FUNCTION subprogram:

Main Program

```
100     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
```

Function Subprogram

```
100     FUNCTION LONG FUN01 (LONG X, LONG Y, LONG Z)
        FUN01 = X + Y + Z
        END FUNCTION
```

These two sets of programs perform essentially the same operation. However, the SUB subprogram both performs the addition and displays the sum, while the FUNCTION subprogram returns a value to the main program, and the main program prints the sum of the variables.

Note that when coding 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.

Note

It is your responsibility to make sure that the data type specified in an EXTERNAL FUNCTION statement matches that specified in the FUNCTION statement.

3.3.2 Parameter Passing Mechanisms

The term *parameter passing mechanism* refers to the way in which data is passed to a subprogram. If a program consists entirely of BASIC program modules, you should not specify any parameter passing mechanisms. This is because the default parameter passing mechanisms are the same for all BASIC programs.

Other programming languages may not pass or receive parameters in the same way BASIC does. Therefore, when calling non–BASIC programs you may have to specify a parameter passing mechanism.

The VAX–11 Procedure Calling Standard allows three methods for passing parameters: 1) immediate value, 2) reference, and 3) descriptor. BASIC supports these methods with the three BY clauses of the CALL statement:

- BY VALUE (immediate value) specifies that BASIC passes actual 32–bit values to the subprogram.
- BY REF (reference) specifies that BASIC passes the address of a storage location to the subprogram.
- BY DESC (descriptor) specifies that BASIC passes the address of a VAX–11 descriptor to the subprogram. These descriptors include pointers to the data. See the *VAX–11 Architecture Handbook* for more information on VAX–11 descriptors.

For example:

```
100     CALL SUB1 BY REF (VAL1, VAL2, VAL3)
200     CALL SUB2 (VAL1 BY REF, VAL2 BY VALUE, VAL3 BY DESC)
300     CALL SUB3 (VAL1 BY VALUE, VAL2, VAL3 BY DESC)
```

Note that a BY clause outside the parameter list applies to all parameters unless overridden by another BY clause inside the parentheses. However, BY clauses inside the parentheses apply only to a single parameter.

Table 3–1 shows allowable parameters mechanisms for each type of parameter. In this table, asterisks indicate the default parameter passing mechanism.

Table 3–1: Allowable Parameter Passing Mechanisms

Parameter	BY VALUE	BY REF	BY DESC
Numeric Data			
Variables	YES	*YES	YES
Constants	YES	*Local copy	Local copy
Expressions	YES	*Local copy	Local copy
Elements of a nonvirtual array	YES	*YES	YES
Virtual array elements	YES	*Local copy	Local copy
Nonvirtual entire array	NO	YES	*YES
Virtual entire array	NO	NO	NO

Table 3–1: Allowable Parameter Passing Mechanisms (Cont.)

Parameter	BY VALUE	BY REF	BY DESC
String Data			
Variables	NO	YES	*YES
Constants	NO	Local copy	*Local copy
Expressions	NO	Local copy	*Local copy
Nonvirtual array elements	NO	YES	*YES
Virtual array elements	NO	Local copy	*Local copy
Nonvirtual entire arrays	NO	YES	*YES
Virtual entire arrays	NO	NO	NO
Packed Decimal Data			
Variables	NO	*YES	YES
Constants	NO	*Local copy	Local copy
Expressions	NO	*Local copy	Local copy
Nonvirtual array elements	NO	*YES	YES
Virtual array elements	NO	*Local copy	Local copy
Nonvirtual entire arrays	NO	YES	*YES
Virtual entire arrays	NO	NO	NO
Other Parameters			
RECORD variables	NO	*YES	YES
RFA variables	NO	*YES	YES

In no case should you use a BY clause when calling a BASIC subprogram from a BASIC main program. The default parameter passing mechanisms for the CALL statement correspond precisely to the way a BASIC subprogram expects to receive the parameters. Use the BY clauses only when the main and subprograms are written in different languages.

If the parameter list has two commas with no expression between them, BASIC passes a null argument in that position. For example:

```
2000 CALL MACRO1 (SS_STRING$ BY DESC, , , 35% BY VALUE)
```

This statement calls a VAX–11 MACRO subprogram specifying:

- The first parameter is a string, passed by descriptor
- The second and third parameters are null
- The fourth parameter is an integer, passed by value

Specifying a null parameter is the same as specifying 0% BY VALUE for that parameter; BASIC places a zero in the argument list. You cannot pass null parameters to a BASIC subprogram, however.

The BY VALUE parameter passing mechanism is recommended only for system services and subprograms written in VAX–11 MACRO and BLISS–32. Further, you should use BY VALUE only for BYTE, WORD, LONG, and SINGLE values because the argument list allows only 32 bits per argument.

3.3.2.1 Local Copies

If a parameter is an expression, a function, or a virtual array element, then it is not possible to pass the parameter's address. In these cases, BASIC makes a local copy of the parameter's value and passes this local copy by reference.

You can force BASIC to make a local copy of any parameter by enclosing it in parentheses. This is a useful technique for parameters that BASIC passes by reference. By forcing BASIC to make a local copy, you make it impossible for the subprogram to modify the actual parameter. For example:

```
100     DECLARE LONG A
        CALL SUB1 ((A))
        PRINT A
        END
200     SUB SUB1 (LONG B)
        B = 3
        END SUB
```

When variable A is printed in the main program, the value is zero because the expression (A) is not modifiable by the subprogram. By removing the extra parentheses from variable A, you allow the subprogram to modify the parameter.

3.3.2.2 Argument Lists

The VAX–11 Procedure Calling Standard defines an argument list as a sequence of longword (32–bit) entries, the first of which contains an argument count. It indicates how many arguments follow in the list. BASIC conforms to this standard.

3.4 BASIC Subprograms

BASIC has both SUB and FUNCTION subprograms. Both types receive parameters and can modify parameters passed by reference. The differences between a SUB subprogram and a FUNCTION subprogram are:

- FUNCTION subprograms must be declared with an EXTERNAL statement in the calling program. Declaring a SUB subprogram is optional.

- FUNCTION subprograms are called by referencing the FUNCTION name. SUB subprograms are called with the CALL statement.

- FUNCTION subprograms return a value; SUB subprograms do not.

Note that either type of subprogram can return a value to the calling program by way of a parameter passed by reference (a modifiable parameter).

BASIC subprograms begin with either: 1) a SUB statement or 2) a FUNCTION statement. You call subprograms of the first kind with the CALL statement. You call subprograms of the second kind by declaring them to be external functions in an EXTERNAL statement. You then use the function subprogram just as you would use any BASIC user-defined function.

You can pass approximately 255 parameters to a subprogram, depending on the complexity of expressions in the parameter list. Separate parameters with commas. The default parameter passing mechanisms for BASIC subprograms are:

- BY REF for all parameters except strings and entire arrays.
- BY DESC for strings and arrays.

Although the BASIC compiler treats each subprogram as a separate unit, you can create and compile a single BASIC source file containing a main program and multiple subprograms.

When you compile a source file containing multiple modules, BASIC creates a single object module, which can then be linked and executed. You can also RUN such a multi-module source file in the BASIC environment. See Section 3.5 for more information.

3.4.1 SUB Subprograms

The SUB statement marks the beginning of a SUB subprogram. Its format is:

```
SUB sub-nam ( [ formal-param ],... ) ]
```

where:

sub-nam	Is a unique, 1– to 31–character name. Neither the subprogram nor main program can have a MAP or a COMMON block of this name.
formal-param	Is a parameter.

The formal parameters in the SUB statement must agree in number and data type with the actual parameters in the calling statement.

The END SUB statement marks the end of a BASIC subprogram. Its format is:

```
END SUB
```

SUBEND is a synonym for END SUB; however, END SUB is the preferred usage. END SUB must be the last statement in a SUB subprogram, and it designates the end of the subprogram. END SUB transfers control to the statement immediately after the statement that called the subprogram. The END SUB statement must be the last statement in the subprogram.

You can also exit from a subprogram with the EXIT SUB statement. Its format is:

```
EXIT SUB
```

SUBEXIT is a synonym for EXIT SUB; however, EXIT SUB is the preferred usage. The EXIT SUB statement is equivalent to an unconditional transfer to the END SUB statement.

Main programs and subprograms can use the same variable names, line numbers, and so forth. However, they can share data only if the data is:

- Part of a MAP or COMMON block
- In a file
- Passed as a parameter in the calling statement

See Chapter 5 in the *BASIC User's Guide* for information about using MAP and COMMON statements.

You can open and access a file in either the main program or subprogram. Because there is a single set of record pointers for all program modules, sequentially accessing a file on a given channel retrieves the next record, whether you perform the access from the main program or subprogram. Similarly, the RESTORE # statement resets the record pointer to the first record in the file whether RESTORE # is executed in the main or subprogram. This example accesses a relative file in a main and subprogram:

Main Program

```
1       ON ERROR GOTO ERR_ROUTINE
        DECLARE INTEGER REC_NUM
        REC_NUM = 0
        MAP (EMPDAT)    LONG    EMP_NUM,                        &
                        STRING  NA_ME = 30,                     &
                                WAGE_CLASS = 2,                 &
                                JOB_TITLE = 20,                 &
                                REVIEW_DATE = 8
        OPEN "EMP.DAT" FOR INPUT AS FILE #6,                    &
                        ORGANIZATION RELATIVE VARIABLE,         &
                        ACCESS MODIFY,                          &
                        MAP EMPDAT
        GET_ROUTINE:
                REC_NUM = REC_NUM + 1
                GET #6, RECORD REC_NUM
                IF WAGE_CLASS = "01"
                THEN
                        CALL REVIEWREP ( REC_NUM )
                        GOTO GET_ROUTINE
                ELSE
                        GOTO GET_ROUTINE
                END IF
                GOTO 1000

        ERR_ROUTINE:
                IF (ERR = 11)
                THEN
                        CLOSE #6
                        PRINT "Finished"
                        RESUME 1000
                ELSE
                        ON ERROR GOTO 0
                END IF

1000    END
```

Subprogram

```
2000    SUB REVIEWREP ( INTEGER N )
                ON ERROR GO BACK
                MAP (EMPDAT) STRING Z = 64
                OPEN "REVREP.DAT" AS FILE #1, ACCESS APPEND
                GET #6, RECORD N
                PRINT #1, Z
                CLOSE #1
        END SUB
```

The main program opens an existing relative file, using the EMPDAT MAP. As the main program retrieves records, it checks the WAGE_CLASS field for a value of "01". If it finds that value, it calls the subprogram REVIEWREP, passing the record number as a parameter.

The REVIEWREP program:

- Receives the record number as a parameter
- Executes an ON ERROR GO BACK, specifying that the main program handles any errors

- Maps EMPDAT as a 64–character string variable
- Opens the terminal-format file REVREP.DAT with ACCESS APPEND

REVIEWREP then retrieves the record specified by the parameter N and prints the variable Z to the terminal format file. Control then returns to the main program so that it can scan for another record.

You can pass string, integer, or floating-point data to a BASIC subprogram. The data can be:

- Constants
- Variables
- Expressions
- Functions
- Array elements
- Entire arrays (but not virtual arrays)

Table 3–1 summarizes allowable argument passing mechanisms for various arguments.

For passing constants, variables, functions, and array elements, you simply name them in the argument list. For example:

```
100 CALL 'SUB01'(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 n commas enclosed in parentheses, where n is the number of array dimensions minus one:

- One-dimensional arrays: array-nam()
  For example: A50()
- Two-dimensional arrays: array-nam(,)
  For example: NA_ME$(,)
- Three-dimensional arrays: array-nam(,,)
  For example: THREE_D(,,)

For example:

Main Program

```
100     DECLARE LONG I,J,K, THREE_D(10,10,10)
200     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

300     CALL SUB EXAMPLE_SUB( THREE_D(,,))
400     END
```

Subprogram:

```
100     SUB EXAMPLE_SUB( LONG X( , , )
        !.
        !.
        !.
1000    END SUB
```

This 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 can be observed when control returns to the main program.

In a subprogram, all variables and arrays are local to the program module, unless they are passed or received as parameters or part of a COMMON or MAP. Similarly, all DATA statements are local to a subprogram. 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 no matter where it is executed). Each time you call a subprogram, BASIC positions the data pointer at the beginning of the subprogram's data.

Note that a DEF function defined in a subprogram is also local to the subprogram.

3.4.2 FUNCTION Subprograms

A FUNCTION subprogram is a program module that can be separately compiled. FUNCTION subprograms return a value. To create the function subprogram, you use the FUNCTION, END FUNCTION, and EXIT FUNCTION statements. The only difference between a FUNCTION subprogram and a SUB subprogram is that the FUNCTION subprogram returns a value.

FUNCTION subprograms are useful because:

- They can be invoked by any program module, not just the one in which they reside
- They allow up to 255 parameters
- Unlike DEF function definitions, they allow array parameters

You use the EXTERNAL statement to name and explicitly declare the data type of an external function.

The FUNCTION statement marks the beginning of a FUNCTION subprogram. Its format is:

FUNCTION data-type func-nam [([formal-param],...)]

where:

data-type	Is any valid BASIC data-type keyword or RECORD name. The data-type keyword following FUNCTION specifies the data-type of the return value. Data-type keywords preceding formal parameters specify the data type of that parameter. See Chapter 6 for more information about the RECORD statement.
func-nam	Is a 1– to 31–character subprogram name.
formal-param	Is a formal parameter.

The END FUNCTION statement: 1) marks the end of a function subprogram, 2) returns a value, and 3) returns program control to the statement that invoked the function. Its format is:

END FUNCTION

FUNCTIONEND is a synonym for END FUNCTION; however, END FUNCTION is the preferred usage.

The EXIT FUNCTION statement immediately returns program control to the statement that invoked the function. It is equivalent to an unconditional transfer to the END FUNCTION statement. Its format is:

```
EXIT FUNCTION
```

FUNCTIONEXIT is a synonym for EXIT FUNCTION; however, EXIT FUNCTION is the preferred usage.

Here is an example of an external function:

```
100     FUNCTION REAL SPHERE_VOLUME (REAL R)
            EXIT FUNCTION IF R <= 0
            SPHERE_VOLUME = 4/3 * PI * R ** 3
        END FUNCTION
```

This function returns the volume of a sphere of radius R. If this function is invoked with an actual parameter value less than or equal to zero, the function returns zero.

This example declares the function subprogram and invokes it:

```
1000    EXTERNAL REAL FUNCTION SPHERE_VOLUME
1100    PRINT SPHERE_VOLUME(5.925)
```

Note that this module is compiled separately from the FUNCTION subprogram. You can link these modules together to run the program from DCL level. To run the program in the BASIC environment, you:

1. Compile the function subprogram
2. Load the resulting object module with the LOAD command
3. Read in the main program with the OLD command
4. Type RUN

See Chapter 2 for more information about the LOAD command.

3.5 Compiling Subprograms

A BASIC source file can contain multiple program modules. When you compile such a file, BASIC produces a single object file containing the code from all the program modules. This object file can be linked to create an executable image.

If the main program and subprograms are in separate source files, you can compile them from DCL level separately, for example:

```
$ BASIC main,sub1,sub2
```

This command causes BASIC to create MAIN.OBJ, SUB1.OBJ, and SUB2.OBJ. To link these programs, you must specify all object files as input to the VAX–11 Linker.

Alternatively, you can compile multiple modules into a single object file from DCL command level, for example:

```
$ BASIC main+sub1+sub2
```

This command causes BASIC to create a single object file named MAIN.OBJ from the three source modules. To link this program, you specify only one input file to the linker. In the BASIC environment, you can compile multiple modules into a single object file only by using the APPEND command followed by the COMPILE command.

When the main program and subprograms are in a single source file:

1. Each line number must be unique.
2. The main program's END statement must have a lower line number than any subprogram's SUB or FUNCTION statement.
3. Any subprogram module must end (with an END SUB or END FUNCTION) before the next subprogram module starts.

This is an example of a combined main program and subprogram:

```
100     EXTERNAL STRING FUNCTION STRIP
             A$ = 'DB0:[33,4]      '
             B$ = STRIP( A$ )
             PRINT B$
500     END
1000    FUNCTION STRING STRIP( STRING ALPHA )
1100    IF (POS( ALPHA, '[', 1%)) > 0
             THEN STRIP = EDIT$(ALPHA, 128% +64%)
             ELSE STRIP = ALPHA
1200    END FUNCTION
```

If you do not assign a value to the function name, the function returns zero or the null string.

3.6 Calling Non–BASIC Subprograms

When calling non–BASIC subprograms, you should always declare them with the EXTERNAL statement. By declaring the non–BASIC subprogram, you specify the data types and parameter passing mechanisms only once. After the subprogram is declared, BASIC automatically converts parameters to the proper data type and specifies the proper parameter passing mechanism for every CALL to the subprogram. For example:

```
100     DECLARE WORD XYZ, STRING ABC
200     EXTERNAL COBSUB (LONG BY VALUE, STRING BY DESC)
        !.
        !.
        !.
5000    CALL COBSUB (XYZ, ABC)
```

Line 100 declares two program variables: a WORD integer XYZ and a dynamic string ABC. The EXTERNAL statement in line 200 declares COBSUB as an external subprogram with two parameters: a LONG integer and a string. When COBSUB is called at line 5000, BASIC automatically passes the value contained in XYZ as a LONG integer because this is the data type specified in the EXTERNAL statement.

Note that in a CALL using parameters passed BY REF, if the actual parameter's data type does not match that specified in the EXTERNAL statement, BASIC reports the compile-time error "PARMODCHA, Mode for parameter of routine changed to match declaration". This tells you that 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. BASIC warns you of this because the change means that the parameter can no longer be modified by the subprogram.

3.7 Calling BASIC Subprograms from Other Languages

When writing a BASIC subprogram to be called from another language, the SUB and FUNCTION statements have the format:

```
SUB sub-name [ pass-mech ] [ ( [ formal-param ],... ) ]

              [ statement ]...

{ END SUB }
{ SUBEND  }


    pass-mech:   { BY DESC }
                 { BY REF  }


    formal-param:          { unsubs-vbl-nam                          }
                [ data-type ] { array-nam ( [ int-const ]  ,... )    }  [ = int-const ] [ pass-mech ]
                           {              [ ,         ]  ...        }
```

This format lets you specify a parameter passing mechanism to match that of the calling program. You should use this format only when the BASIC subprogram is being called by a program written in a language other than BASIC. Note that the only supported non-default parameter passing mechanism in the SUB statement is BY REF for strings and arrays.

Note

> Because VAX–11 languages conform to a calling standard, it is better programming practice to specify a parameter passing mechanism in the calling program than in the receiving program.

VAX–11 FORTRAN passes and receives numeric data by reference; the default parameter passing mechanisms are all that is required for passing numeric data back and forth between FORTRAN and BASIC programs.

Both 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:

- By "pre-extending" the string, that is, setting the string variable equal to SPACE$(n), where n is large enough to receive the result

- By defining the string as fixed-length, that is, by naming it in a COMMON 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, because this is the way the FORTRAN subprogram expects to receive it.

You can pass only the data types that BASIC and FORTRAN have in common. Thus, you cannot pass a complex number from a FORTRAN program to a BASIC program, because BASIC does not support complex numbers. However, you could pass a complex number as two floating-point numbers and deal with them independently in the BASIC program.

Chapter 4
The VAX–11 Symbolic Debugger

The VAX–11 Symbolic Debugger lets you debug programs by monitoring the flow of program execution and logic. For a complete description of debugger capabilities, see the *VAX–11 Symbolic Debugger Reference Manual.*

The debugger lets you:

- Set breakpoints (stop program execution just before a specified instruction is executed)
- Set tracepoints (have the debugger pause and display a message whenever a specified instruction or a specified type of instruction is executed)
- Set watchpoints (have the debugger stop and display a message whenever a specified location is modified)
- Examine and modify instructions and data
- Evaluate expressions, perform radix conversions, and compute address values
- Step through a program (tell the debugger to execute one or more instructions and then display a message)

The debugger needs information generated by both the BASIC compiler and the linker. Specifying the /DEBUG qualifier for the COMPILE command or the BASIC DCL command creates the symbolic information for the debugger. Specifying the /DEBUG qualifier for the LINK DCL command makes this information available to the debugger. BASIC supports the following options at compile time for /DEBUG=options: ALL, NONE, [NO]TRACEBACK, and [NO]SYMBOLS.

If you omit the /DEBUG qualifiers for the COMPILE and LINK commands, you can specify /DEBUG for the RUN DCL command. In this case, no symbolic information is available to the debugger; every reference to a program variable or location must be in terms of its absolute address.

If you do not specify /DEBUG for any of the COMPILE, LINK, or RUN commands, and an error occurs, you receive a traceback list (a description of the logic flow to the point that caused the error). However, you cannot invoke the debugger. (If you link your program with the /NOTRACE qualifier, you do not receive the traceback list.)

Table 4–1 lists debugger commands and keywords and their abbreviations.

Table 4–1: Debugger Commands and Keywords

Command Names	Keywords
SET (SE)	LANGUAGE (LA)
SHOW (SH)	MODULE (MODU)
CANCEL (CAN)	SCOPE (SC)
EXAMINE (E)	BREAK (B)
EVALUATE (EV)	TRACE (T)
DEPOSIT (D)	WATCH (W)
DEFINE (DEF)	
EXIT (EXI)	
STEP (S)	
GO (G)	
CALL (CA)	

4.1 Debugger Symbol Table

The debugger maintains a table of symbols defined by the program with which it is linked. This table provides the name of each symbol defined in the program, its data type, and its address. The table also provides information for array boundaries and length information for string data.

The debugger's active symbol table provides room for approximately 2000 symbols. Thus, you should keep track of the number of symbols defined in the programs you are debugging. If your program contains more than one program unit, use the SET MODULE command to be sure the symbol table contains symbols from the program units you wish to debug. Use the CANCEL MODULE command to remove symbols defined in program units that no longer need debugging. The SET MODULE and CANCEL MODULE commands are defined in Section 4.2.2.

4.2 Preparing to Debug a Program

The following sections describe the commands used to establish the proper environment for debugging BASIC programs. These commands are:

SET LANGUAGE
SHOW LANGUAGE

SET MODULE
SHOW MODULE
CANCEL MODULE

SET SCOPE
SHOW SCOPE
CANCEL SCOPE

You can use these commands to change the initial settings for module, language, and scope. The initial settings for entry and display are:

SYMBOL
DECIMAL

4.2.1 SET LANGUAGE and SHOW LANGUAGE Commands

The SET LANGUAGE command tells the debugger that the debugging dialog is to be conducted according to the conventions of the specified language. For example, if you specify SET LANGUAGE BASIC, the debugger will accept and display numeric values in decimal radix.

The command has the form:

SET LANGUAGE language

where:

language   Specifies the language to be used.

To determine which language is currently in effect, use the SHOW LANGUAGE command. This command has the form:

SHOW LANGUAGE

The debugger responds by displaying the language in effect. For example:

```
DBG>SHOW LANGUAGE
language: BASIC
```

4.2.2 SET MODULE, SHOW MODULE, and CANCEL MODULE Commands

The module commands let you control the contents of the debugger's active symbol table when the program you want to debug consists of multiple program units. These commands perform the following functions:

- SET MODULE places the symbols defined in the specified program unit into the active symbol table. If you do not specify a module, the active symbol table includes all global symbols and local symbols for the first program unit specified in the LINK command.
- SHOW MODULE displays the names of all program units whose symbols are potentially available. "Yes" means the symbols for that module are available; "no" means they are not.
- CANCEL MODULE removes the specified program unit's symbols from the active symbol table.

The SET MODULE command has the form:

SET MODULE { program-unit [,program-unit] ,... | /ALL }

where:

program-unit   Specifies the name of the program unit whose symbols are to be included in the active symbol table.

/ALL   Requests the debugger to set the symbols of all known modules. If there is insufficient space, the debugger displays an error message.

The SHOW MODULE command has the form:

SHOW MODULE

This command takes no parameters. The debugger responds by displaying the names of the modules linked with the debugger, indicating the modules whose symbols are included in the image and their sizes. Only those module names marked "Yes" have their symbols in the active symbol table.

The CANCEL MODULE command has the form:

```
CANCEL MODULE   { program-unit [,program-unit] ... }
                { /ALL                             }
```

where:

program-unit	Specifies the name of the program unit for which symbols are to be removed.
/ALL	Specifies that all information is to be purged from the active symbol table.

4.2.3 SET SCOPE, SHOW SCOPE, and CANCEL SCOPE Commands

The SCOPE commands let you control the default used to resolve references to symbols. When you execute a program under the debugger's control, the debugger loads the program and issues the DBG> prompt. At this point the current scope is the module containing the program's first instruction. If you display the current scope, the debugger prints:

```
scope: 0 [ = <module>\<module> ]
```

This tells you that the scope is set to the current program module and gives the module's name.

When you use a command such as EXAMINE, you can either specify or omit the name of the module in which the symbol is defined. If you omit the module name, the debugger first checks the current SCOPE. If it fails to resolve the symbol, it searches other modules for a unique variable of the given name. If it still cannot resolve the symbol, the debugger displays a message.

The SCOPE commands perform the following functions:

- SET SCOPE defines the specified program unit(s) to be the default.
- SHOW SCOPE displays the current default scope.
- CANCEL SCOPE revokes the default program unit(s) named previously in a SET SCOPE command.

The SET SCOPE command has the form:

SET SCOPE program-unit [, program unit. . .]

where:

program-unit	Specifies the name of the program unit to be used as the default.

For example:

```
SET SCOPE MAXI, MINI
```

This command tells the debugger to search program units MAXI and MINI (in that order) to resolve a symbol.

The SHOW SCOPE command has the form:

```
SHOW SCOPE
```

This command takes no parameters. The symbol displayed indicates the current SCOPE.

The CANCEL SCOPE command has the form:

```
CANCEL SCOPE
```

This command takes no parameters. SCOPE becomes the current program unit.

4.3 Controlling Program Execution

To see what happens during execution of your program, you must be able to suspend and resume the program at specific points. The following commands are available for these purposes:

SET BREAK
SHOW BREAK
CANCEL BREAK
SET TRACE
SHOW TRACE
CANCEL TRACE
SET WATCH
SHOW WATCH
CANCEL WATCH

SHOW CALLS

GO
STEP

CTRL/Y

EXIT

4.3.1 SET BREAK, SHOW BREAK, and CANCEL BREAK Commands

The BREAK commands let you select specific locations for program suspension, so that you can examine or modify variables or arrays in the program. The BREAK commands perform the following functions:

- SET BREAK defines an address or line number and statement number at which to suspend execution.
- SHOW BREAK displays all breakpoints currently set in the program.
- CANCEL BREAK removes selected breakpoints.

The SET BREAK command has the form:

```
SET BREAK [/AFTER:n]   { %LINE lin-num[.stmnt-num] }   [DO(commands(s))]
                       { address                   }
```

where:

lin-num[.stmnt-num] Specifies the line and statement at which the breakpoint is to occur. To specify a single statement on a multi-statement line, follow the line number

with a decimal point and the statement number. (For example, 20.4 is the fourth statement on line 20.) The numbering of statements is indicated on the program listing produced by specifying /LIST. Execution is suspended just before the specified location.

address — Specifies a storage location.

DO(commands(s)) — Requests that the debugger perform the specified commands, if any, when the breakpoint is reached.

For example:

```
SET BREAK %LINE 100.2 DO(EXAMINE TOTAL; EXAMINE AREA)
```

This command examines variables TOTAL and AREA when the breakpoint at the second statement on line 100 is reached.

You can use the /AFTER qualifier to control when a breakpoint takes effect. Thus, if you set a breakpoint on a line that is in a FOR/NEXT loop, and you want the breakpoint to be effective the third time through the loop, then specify the /AFTER switch:

```
DBG>SET BREAK/AFTER:3 %LINE 20
```

Note that if you use the /AFTER qualifier, the breakpoint is reported the nth time it is encountered, and every time it is encountered thereafter.

The SHOW BREAK command has the form:

SHOW BREAK

This command takes no parameters. The debugger responds by displaying the location of breakpoints.

The CANCEL BREAK command has the form:

```
              ⎧ %LINE lin-num[.stmnt-num] ⎫
CANCEL BREAK  ⎨  address                  ⎬
              ⎩ /ALL                      ⎭
```

where:

lin-num[.stmnt-num] — Removes the breakpoint at the specified line and statement.

address — Removes the breakpoint at the specified address.

/ALL — Removes all breakpoints in the program.

4.3.2 SET TRACE, SHOW TRACE, and CANCEL TRACE Commands

The TRACE commands let you set, examine, and remove tracepoints in your program. A tracepoint is similar to a breakpoint in that it suspends program execution and displays the address at the point of suspension. However, program execution resumes immediately. Thus, tracepoints let you follow the sequence of program execution to ensure that execution is being carried out in the proper order.

Tracepoints and breakpoints are mutually exclusive. If you set a tracepoint at the same location as a current breakpoint, the breakpoint will be canceled, and vice versa.

The TRACE commands perform the following functions:

- SET TRACE establishes points within the program at which execution is momentarily suspended.
- SHOW TRACE displays the locations in the program at which tracepoints are currently set.
- CANCEL TRACE removes one or more tracepoints currently set in the program.

The SET TRACE command has the form:

```
SET TRACE    { %LINE lin-num[.stmnt-num] }
             {  address                  }
```

where:

lin-num[.stmnt-num]	Specifies the line and statement at which the tracepoint is to occur.
address	Specifies the address at which the tracepoint is to occur.

The SHOW TRACE command has the form:

```
SHOW TRACE
```

This command takes no parameters.

The CANCEL TRACE command has the form:

```
               { %LINE lin-num[.stmnt-num] }
CANCEL TRACE   {  address                  }
               { /ALL                      }
```

where:

lin-num[.stmnt-num]	Removes the tracepoint at the specified line and statement.
address	Removes the tracepoint at the specified address.
/ALL	Removes all tracepoints in the program.

4.3.3 SET WATCH, SHOW WATCH, and CANCEL WATCH Commands

The WATCH commands let you monitor specified locations to determine when attempts are made to modify their contents and take the appropriate actions. These locations are called watchpoints. When an attempt is made to change a watchpoint, the debugger halts program execution, displays the address of the instruction, and prompts for a command. Watchpoints are monitored continuously. Thus, you can determine whether locations are being modified inadvertently during program execution.

Note

You can set watchpoints only on variables declared in COMMONs or MAPs.

The WATCH commands perform the following functions:

- SET WATCH defines the location(s) to be monitored.
- SHOW WATCH displays the locations currently being monitored.
- CANCEL WATCH disables monitoring of specified locations.

The SET WATCH command has the form:

SET WATCH vbl

where:

vbl Specifies the location to be monitored. You can monitor scalar variables and array elements.

For example:

```
SET WATCH AREA
```

Note that watchpoints, tracepoints, and breakpoints are mutually exclusive.

The SHOW WATCH command has the form:

SHOW WATCH

This command takes no parameters. All watchpoints are displayed.

The CANCEL WATCH command has the form:

CANCEL WATCH { vbl | /ALL }

where:

vbl Specifies the location for which monitoring is to be disabled.

/ALL Removes all watchpoints from the program.

For example:

```
CANCEL WATCH AREA
```

4.3.4 SHOW CALLS Command

This command can be used to produce a traceback of calls, and is particularly useful when you have returned to the debugger following a CTRL/Y command. It has the form:

SHOW CALLS [n]

The debugger displays a traceback list, showing the sequence of calls leading to the current module. If you include a value for n, the n most recent calls are displayed.

4.3.5 GO and STEP Commands

These commands let you initiate and resume program execution.

- GO initiates execution at a specified location and continues to the end of the program or to the next breakpoint.
- STEP initiates execution from the current location and continues for a specified number of statements.

The form of the GO command is:

```
GO    { %LINE lin-num[.stmnt-num] }
      { address                   }
```

where:

lin-num[.stmnt-num]	Specifies the line and statement at which execution is to begin.
address	Specifies the address at which program execution is to begin. The address parameter is optional; if you omit it, execution starts at the current location.

Note

You cannot restart a program from the beginning unless you first exit from the debugger. If you try to restart from the beginning, the results are undefined.

The form of the STEP command is:

STEP [/qualifiers] [n]

The value specified for n determines the number of statements to be executed. If you specify 0, or omit n, a default of 1 is assumed. Note, however, that if you issue a STEP command while your program is stopped in a module whose symbols are not set in the active symbol table, then n instructions (not statements) will be executed.

You can specify the following qualifiers for the STEP command:

/[NO]SYSTEM
/OVER
/INTO
/LINE
/INSTRUCTION

where:

/[NO]SYSTEM	/SYSTEM tells the debugger to count steps wherever they occur, including system address space. The default is /NOSYSTEM.
/OVER	Tells the debugger to ignore calls to subprograms as it steps through the program. That is, it steps over the call. This is the default.
/INTO	Tells the debugger to recognize calls to subprograms as it steps through the program. That is, it is requested to step into the subprogram.
/LINE	Tells the debugger to step through the program on a line-by-line basis (default for BASIC).
/INSTRUCTION	Tells the debugger to step through the program on an instruction-by-instruction basis.

You can specify these qualifiers each time you issue a STEP command, or you can use a SET STEP command, as shown in the following example:

```
SET STEP INSTRUCTION, INTO, SYSTEM
```

This command specifies that all defaults applicable to BASIC programs are to be overridden. When you subsequently issue a STEP command with no qualifiers, these qualifiers are assumed to be in effect. You can, however, supersede them by including a qualifier with a STEP command. For example:

```
STEP/LINE 10
```

This command tells the debugger to execute 10 lines, regardless of the SET STEP command.

It is advisable to use STEP to execute only a few instructions at a time. To execute many instructions and then stop, use a SET BREAK command to set a breakpoint and then issue a GO command.

4.3.6 CTRL/Y Command

You can use the CTRL/Y command at any time to return to the system command level. This command is issued when you press the CTRL key and the Y key at the same time. The $ prompt will be displayed on the terminal. To return to the debugger, type DEBUG. You can use the CTRL/Y command if your program loops or otherwise fails to stop at a breakpoint. To find out where you were at the instant CTRL/Y was executed, use the SHOW CALLS command after you return to the debugger. See Section 4.3.4.

4.3.7 EXIT Command

The EXIT command lets you exit from the debugger when you are ready to terminate a debugging session. It has the form:

EXIT

This command takes no parameters. You must use the EXIT command when your program terminates to return to system command level.

4.4 Examining and Modifying Locations

Once you have set breakpoints and begun execution, the next step is to see whether correct values are being generated and, possibly, to change the contents of locations as execution proceeds. You may also want to calculate the value of an expression that appears in your program. The debugger provides the following commands for these purposes:

EXAMINE

DEPOSIT

EVALUATE

4.4.1 EXAMINE Command

The EXAMINE command lets you look at the contents of specified locations. It has the form:

```
EXAMINE  { vbl [,vbl]             }
         { address[:address]      }
```

where:

vbl Specifies a simple or subscripted variable.

address Specifies the address whose contents are to be examined. If you specify two addresses separated by a colon, the debugger displays the contents of all addresses in that range.

For example:

```
EXAMINE IZZY
```

The contents of variable IZZY are displayed.

```
EXAMINE IARR(I)
```

The contents of the Ith element in array IARR are displayed.

```
EXAMINE IARR(I):IARR(10)
```

The contents of the first through tenth elements of the array IARR are examined.

You can also display the contents of an absolute address. For example:

```
EXAMINE 600
```

The contents of absolute address 600 are displayed.

You can also examine the contents of a register by using EXAMINE with the register name (for example, R1, R2, PC, and SP):

```
EXAMINE R1
```

If the register is being used to store data, EXAMINE R1 displays the data currently in the register. If the register is being used as a pointer, EXAMINE R1 displays the absolute address of the data. You can then use a second EXAMINE command with the absolute address to display the actual data.

If LANGUAGE is set to BASIC, you cannot use an indirect "contents of" operator (@ or .).

4.4.2 DEPOSIT Command

The DEPOSIT command lets you change the contents of specified locations. It has the form:

```
DEPOSIT { vbl=value                    }
        { address=value[,value ... ]   }
```

where:

vbl Specifies the variable into which the value is deposited.

address Specifies the address into which the value is deposited.

value Specifies the value to be deposited.

You can change the contents of a specific location or of several consecutive locations, as shown in the following examples.

```
DEPOSIT IZZY=100
```

This command places the decimal value 100 into the variable IZZY.

```
DEPOSIT IARR(1)=100, 150, 200
```

This command places the decimal values 100, 150, and 200 into elements 1, 2, and 3 of array IARR.

The delimiters used to enclose ASCII strings in the DEPOSIT command can be either apostrophes (') or quotation marks ('').

4.4.3 EVALUATE Command

The EVALUATE command lets you use the debugger as a calculator, to determine the value of expressions. It has the form:

EVALUATE expression

where:

expression Specifies the expression whose value is to be determined.

For example:

```
EVALUATE A(J) * I
```

The value of this expression will be displayed. You can also use the EVALUATE command to determine addresses, as follows:

```
EVALUATE/ADDRESS expression
```

For example:

```
EVALUATE/ADDRESS I
```

This calculates the address of the variable I, in decimal.

```
EVALUATE/ADDRESS A(J)
```

This calculates the address of the Jth element of array A.

You can also use EVALUATE to perform address arithmetic, such as computing an offset or array element address. For example:

```
EVALUATE/ADDRESS I+4
```

4.4.4 Specifying Address

The debugger allows you to express addresses in symbolic form. Thus, to examine a location, you need only refer to it by its symbolic name. You don't have to concern yourself with its location in

memory unless you omitted the /DEBUG qualifier from the BASIC and LINK commands. Simply specify the variable, array element, or function name in the debugger command.

4.4.4.1 Specifying Scope

If the program you are debugging consists of more than one program module, you must be sure that your symbol references are unambiguous. For example, if your main program calls a subroutine and the symbols from both program units are in the debugger's symbol table, you must distinguish between duplicate symbols.

For example, assume that you want to set a breakpoint in the subroutine, and you issue the following command:

```
SET BREAK %LINE 10
```

Because you do not specify a program unit name in this command, the debugger uses a default to decide which line 10 you mean. If you used a SET SCOPE command, the debugger uses the program unit specified in the SET SCOPE command. To override this default, you must specify a command in the following general form:

```
SET BREAK %LINE program-unit\10
```

For example:

```
SET BREAK %LINE ARGO\10
```

This command specifically calls for a breakpoint to be set at line 10 in the program unit named ARGO.

Unambiguous references are also required when you specify variables. If there are duplicate variable names (for instance, X) you should specify which X you want. For example:

```
EXAMINE SUB3\X
```

This command specifies the variable X in module SUB3.

4.4.4.2 Previous, Current, and Next Locations

The debugger provides a quick method for referring to any of three locations:

- The previous location
- The current location
- The location at the next higher address (next location)

For BASIC programs, you should refer to the previous and next locations only for MAP, COMMON, or array variables. This is because these types of variables are in static storage. The debugger will let you refer to the previous and next locations for dynamic variables, but these locations do not hold useful information.

To specify the previous location, type a circumflex (^). For example:

```
EXAMINE ^
```

This command displays the contents of the previous location.

To specify the current location, type a period (.). For example:

```
DEPOSIT .=100
```

This command puts a decimal value of 100 in the current location. This method is most useful after you have looked at a location and decided to change it or when you want to verify that a DEPOSIT command has been executed as expected.

To specify the next higher location, simply omit the address value entirely. For example:

```
EXAMINE
```

The next location's contents will be displayed.

4.4.4.3 Defining Addresses Symbolically

You may occasionally need to access absolute addresses. To help you do so, the debugger provides the DEFINE command, which creates a symbolic reference for an absolute address. Then you can refer to the address by its symbolic name, rather than by its absolute value. The DEFINE command has the form:

DEFINE name = address

For example:

```
DEFINE TOP=1036
```

Subsequent references to this address can be made using the symbol TOP. For example:

```
DEPOSIT TOP=256
```

The contents of address 1036 will be changed to 256.

4.4.5 Calling Subroutines from the Debugger

The CALL command lets you call a subroutine from the debugger. It has the form:

CALL sub [(param, ...)]

where:

sub Specifies the subroutine name.

param Specifies one or more actual arguments.

Control returns to the debugger from the subroutine at the point at which the CALL command was issued. The context that existed at the time of the CALL is also restored.

4.4.6 Debugger Command Qualifiers

Qualifiers can be used to modify some debugging commands. The general form in which qualifiers are specified is:

command / qualifier

Qualifiers change the defaults the debugger uses in processing commands. For example, when you deposit a value, the debugger uses decimal radix by default. You can override the default by specifying either /HEX or /OCT. Table 4–2 summarizes the command qualifiers of particular significance in BASIC debugging.

Table 4–2: Debugger Command Qualifiers

Qualifier	Function	Command
/ADDRESS	Indicates that an address value is desired	EVALUATE
/HEX /OCT	Overrides the default radix (decimal)	EVALUATE DEPOSIT

Refer to the *VAX–11 Symbolic Debugger Reference Manual* for more information on qualifiers.

4.4.7 Numeric Data Types

The debugger supports all numeric data types used in VAX–11 BASIC. If you try to deposit a numeric value into a variable or array element that does not have a matching data type, the value is converted to the data type of the variable or array element.

When you deposit real numbers, you must specify a decimal point. To distinguish single-precision and double-precision numbers, use E and D, respectively. For example:

Number	Data Type
24.1	Single precision (default)
24.1E0	Single precision
24.1D0	Double precision
241E0	Invalid (no decimal point)

Note that there is no way to distinguish GFLOAT and HFLOAT values. The most precise floating-point number that can be deposited is a double-precision value.

4.4.8 Evaluating Named Constants

To display the value of named constants, you must first use the debugger command:

```
SET LANGUAGE PLI
```

When the language is set to BASIC, the EXAMINE and EVALUATE commands are synonymous. When the language is set to PL/I, the EXAMINE command argument is always interpreted as an address. Therefore, you must use the EVALUATE command to display the value of declared constants. For example:

```
DBG>SET LANGUAGE PLI
DBG>EVALUATE H_CON
    3.00000000000000000000000000000000E+4000
DBG>EVALUATE G_CON
    5.123456789101234E+33
```

Chapter 5
System Services and Run-Time Library Procedures

This chapter describes the use of VAX/VMS System Services and VAX–11 Run-Time Library Procedures in BASIC programs.

System services are procedures that the VAX/VMS operating system uses to:

- Control resource sharing
- Provide for communication between processes
- Perform operating system functions such as I/O operations

Most system services are used primarily by the VAX/VMS operating system on behalf of users. However, many system services are useful for application programming.

The use of some system services is restricted to protect system performance and the integrity of user processes. The privileges and quotas assigned in the User Authorization File determine whether you can use a restricted system service. These privileges and quotas apply to every image that your process executes. For a complete description of available system services and the privileges required to use them, see the *VAX–11 System Services Reference Manual*.

The VAX–11 Run-Time Library (RTL) contains sets of general purpose and language support procedures. An RTL procedure requires a parameter (or argument) list and can return a function value or a completion status.

For a complete description of available RTL procedures see the *VAX–11 Run-Time Library Reference Manual*.

This chapter provides an overview of system services and RTL routines, then explains the programming techniques you need to use them. Both the system services and RTL sections show sample programs and explanations of selected routines.

5.1 System Services

In BASIC, you use the EXTERNAL statement to define system services as external functions. Once you define a system service as an external function, you can invoke it a function reference. System services return information in two ways:

- Output parameters
- Status codes

An output parameter is a variable passed by reference in the system service parameter list; if the system service completes successfully, it places a value in this variable's storage location. Normally, you code a system service as a function reference because system services also return status codes. This status code is the value the function returns, and it describes the success or failure of the operation. If you are not interested in the status code, you can call a system service. However, it is good programming practice to test the returned status code.

5.1.1 VAX/VMS Symbolic Constants

Symbolic constants are names, or symbols, with which values are associated. 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. 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: 1) the symbolic constant SS$_NORMAL or 2) the integer value one. The program would execute the same way in either case. In the first case, the value for SS$_NORMAL is supplied at link time by the VAX–11 Linker. In the second case, the value one is hard-coded into the program as a literal constant. The advantages to using symbolic constants are that:

- Because symbolic constant names are mnemonic, the program is easier to read and understand.
- It is easier to code the symbolic constant and let the linker fill in the value, rather than looking up the value associated with the symbol and hard-coding that value into the program.
- Should the value associated with a symbol ever change, you must relink the program. To change a hard-coded constant, you must edit the source file, then recompile and relink.

The definitions for symbolic constants used by system services are located in the default system library, STARLET.

5.1.2 Testing for Success or Failure (System Status Codes)

When invoked as an external function, system services always return a longword integer describing the status of the operation. The severity level of the return status is contained in the three low-order bits of the status code. The severity level tells you whether or not the system service succeeded. Table 5–1 shows the values and meanings of status codes.

Table 5–1: System Service Status Codes

Low-Order Bits	Severity Level	Code
0 0 0	Warning	W
0 0 1	Success	S
0 1 0	Error	E
0 1 1	Information	I
1 0 0	Severe Error	F
1 1 0	Undefined	–
1 1 1	Undefined	–

The system never returns the bit patterns described as undefined. Therefore, if bit 0 is set in the returned status code, the severity level is either success or information and the operation succeeded.

Thus, you can test for the success or failure of a system service call by testing these bits. In fact, if you are interested only in whether the operation succeeded, you need only test the least significant bit of the status code; if bit 0 is set, the operation succeeded. You perform this test with the BASIC IF statement and a technique called masking.

A mask is the binary representation of an integer (a bit pattern). When used with a logical operator and another integer, it removes or masks all but the bits you want to evaluate. In most cases, the logical operator is AND. For example, suppose you are working with a word-length integer and that you want to know whether bit 0 is set. (Bits are counted from right to left.) You might set up the mask this way:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

This mask is the binary representation of the integer one. You can test whether bit 0 is set in another integer by using the logical operator AND. The AND operator compares two values, bit by bit, and returns a third value. A bit is set in the returned value only if that bit is set in both the mask and the value being tested.

For example, this program uses the value one as a mask to test whether the value 15 has bit 0 set:

```
10 TEST% = 15%
             IF (TEST% AND 1%) = 0%
                 THEN PRINT "BIT ZERO IS CLEAR"
                 ELSE PRINT "BIT ZERO IS SET"
             END IF
   END


RUNNH


BIT ZERO IS SET
```

The internal representation of the logical operation looks like this:

```
1%       0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1

         AND

TEST%    0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1

Result:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
```

Because an AND operation sets a particular bit in the result only if the same bit is set in both the mask and the tested value, and you know that only bit 0 is set in the mask, a nonzero result means that bit 0 is set in the tested value.

Note that SS$_NORMAL is not the only status code whose low-order bits translate as a successful completion. Other status codes return information in addition to success or failure. For example, the status code SS$_BUFFEROVF (which is issued when a character string returned by a system service is longer than the buffer provided to receive it) is also a success code.

5.1.3 Declaring System Services and Symbolic Status Codes: EXTERNAL Statement

The EXTERNAL statement enables you to access any system service as if it were a BASIC function. When declaring a system service, its format is:

```
                                                     [ BY REF   ]
EXTERNAL LONG FUNCTION ss-name ([data-type           [ BY VALUE ]     [,...])
                                                     [ BY DESC  ]
```

where:

LONG	Specifies that the system service returns a longword integer status code.
FUNCTION	Specifies that the symbol is accessed as a function.
ss-name	Is the name of the system service.
BY VALUE BY REF BY DESC	Specify a parameter passing mechanism.

The system service you access can require one or more parameters. When declaring the system service with the EXTERNAL statement, you specify the data type and parameter passing mechanism for each parameter in the function parameter list. When you invoke the system service, you specify only the actual parameters.

It is important to note that system services have no optional parameters. If a system service specifies three parameters, you must supply exactly three parameters when you invoke it. However, you can specify null parameters to accept the system service default. You specify null parameters with a comma in the parameter list.

If the BASIC default parameter passing mechanism differs from the required system service parameter passing mechanism, you must override the default with the appropriate BY clause.

See Chapter 3 for a discussion of external declarations and parameter passing mechanisms.

5.1.4 System Service Examples

This section contains representative examples of system services. You can find required parameters and defaults in the *VAX–11 System Service Reference Manual*.

Any system service that you access as a function returns a longword integer value describing the status of the operation. Therefore, if you access a system service as an external function, you must supply a LONG variable to receive the returned status.

Note

In all examples in this section, the status code is assigned to a variable called SYS_STATUS. If this value is needed as a parameter to another system service call, it is copied into a temporary variable called SAVE_STATUS. SAVE_STATUS is then passed as a parameter.

5.1.4.1 Creating a Mailbox: SYS$CREMBX

It is often useful to exchange data between programs (for example, to synchronize execution or to send messages). A mailbox is a record-oriented pseudodevice that allows data to be passed from one program to another. You create mailboxes with the SYS$CREMBX system service.

The following example creates a mailbox, sends a message, and waits for a reply. Note that another program is required to read messages from and write replies to the mailbox.

```
100     EXTERNAL INTEGER FUNCTION SYS$CREMBX
        EXTERNAL LONG CONSTANT SS$_NORMAL
        DECLARE LONG SYS_STATUS, CHAN
200     SYS_STATUS = SYS$CREMBX(,CHAN,,,,,"LINK01")
        IF (SYS_STATUS AND SS$_NORMAL) = 0%
        THEN PRINT "Error from mailbox creation:"; SYS_STATUS
        ELSE
            OPEN "LINK01" FOR INPUT AS FILE #CHAN
            PRINT #1%, "Hello, this is a test."
            LINPUT #1%, MSG$
            PRINT MSG$, "was received"
        END IF
        END
```

- Line 100 defines SYS$CREMBX as an EXTERNAL FUNCTION, SS$_NORMAL as an EXTERNAL CONSTANT then declares the storage for SYS_STATUS (which receives the returned status code) and CHAN (which receives the returned channel number).
- Line 200 invokes SYS$CREMBX, passes parameters to it, and assigns the returned status code to SYS_STATUS. SYS$CREMBX requires seven parameters:
  - A flag specifying whether the mailbox is permanent or temporary. The default is a temporary mailbox, and the first comma (null argument) accepts this default.
  - The address of a longword integer variable to receive the channel number assigned. You need only specify the variable; the default parameter passing mechanism for integer variables is BY REF.
  - A number specifying the maximum message size in bytes. The third comma (null argument) accepts the system default.
  - The number of bytes of dynamic memory to use for buffering I/O to the mailbox. The fourth comma (null argument) accepts the system default.
  - The protection mask for the mailbox. The fifth comma (null argument) accepts the system default.
  - The access mode associated with the mailbox channel. The sixth comma (null argument) accepts the system default.
  - The address of a string descriptor specifying the mailbox's logical name. Because BASIC's default parameter passing mechanism for strings is BY DESC, you specify only the string literal.

The program then uses a mask of one to test the SYS_STATUS variable for successful completion.

- The program opens the mailbox, then writes a record to it. The program waits at this point until another program reads the record from the mailbox and writes a record to it. If there is no record in the mailbox when the LINPUT # statement executes, the program waits until a record is written there by another program.

5.1.4.2 Translating a Logical Name: SYS$TRNLOG

SYS$TRNLOG translates a logical name to an equivalence name. It places the equivalence name string into a string variable you name 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 two ways:

- Define the string variable's length in a MAP or COMMON.
- 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.

For example:

```
100     EXTERNAL LONG FUNCTION SYS$TRNLOG
        EXTERNAL LONG CONSTANT SS$_NORMAL, &
                    SS$_NOTRAN, &
                    SS$_ACCVIO, &
                    SS$_IVLOGNAM, &
                    SS$_RESULTOVF
        DECLARE LONG SYS_STATUS, WORD RSN_LEN
        DECLARE STRING L_NAME
        COMMON (BUFCOM) STRING RSN_BUF = 80, MSG = 80
        LINPUT "Logical name for translation"; L_NAME

200     SYS_STATUS = SYS$TRNLOG(L_NAME, RSN_LEN, RSN_BUF,,,,)

        SELECT SYS_STATUS

        CASE SS$_NORMAL
            PRINT "Equivalence name is: "; SEG$(RSN_BUF,1,RSN_LEN)

        CASE SS$_NOTRAN
            PRINT "No translation for logical name"

        CASE SS$_ACCVIO
            PRINT "Error when trying to read or write"
            PRINT "the logical name you provided."

        CASE SS$_IVLOGNAM
            PRINT "Invalid logical name"

        CASE SS$_RESULTOVF
            PRINT "Returned string too large for"
            PRINT "receiving buffer."

        CASE ELSE
            PRINT "Unknown error number";SYS_STATUS

        END SELECT

        END
```

Line 100 starts with the necessary external and internal declarations and ends with a LINPUT statement prompting for a logical name to translate. Thus it declares the SYS$TRNLOG routine to be an EXTERNAL FUNCTION returning a longword status code, then goes on to declare SS$_NORMAL, SS$_NOTRAN, SS$_ACCVIO, SS$_IVLOGNAM, and SS$_RESULTOVF to be longword EXTERNAL CONSTANTS. The values of these symbolic constants are the possible return values of the system service.

The line then declares local storage for the SYS_STATUS variable that receives the returned status code, a WORD variable called RSN_LEN which receives the length of the returned string, and STRING variable called L_NAME. The COMMON statement declares an 80–character fixed-length string to receive the returned equivalence name.

Line 200 invokes the SYS$TRNLOG, passing six parameters to it. SYS$TRNLOG returns a status code to SYS_STATUS. The six parameters are:

- The address of the string descriptor pointing to the logical name. Because the BASIC default parameter passing mechanism for strings is BY DESC, you specify only the string variable name, L_NAME.
- The address of a WORD integer variable to receive the length of the returned string. Because the BASIC default parameter passing mechanism for integer variables is BY REF, you need specify only the integer variable, RSN_LEN.
- The address of the string descriptor pointing to the storage location that receives the resulting equivalence name. Because the BASIC default parameter passing mechanism for strings is BY DESC, you need only specify the string variable, RSN_BUF.
- The address of a longword variable to receive the number of the logical name table where the match was found. If you need this data, specify an integer variable to receive it. Because this program specifies a null argument, SYS$TRNLOG does not return the data.
- The address of a longword variable to receive the access mode from which the logical name entry was made. If you need this data, specify an integer variable to receive it. Because this program specifies a null argument, the SYS$TRNLOG does not return the data.
- A mask specifying which logical name tables are searched. By default, the system searches all tables.

The program then uses the SELECT statement to test the value of the code returned in SYS_STATUS. Each CASE block specifies one of the possible symbolic return values and the actions to be taken based on this value. The CASE ELSE block handles any unexpected errors.

5.1.4.3 Translating System Status Codes: SYS$GETMSG

SYS$GETMSG translates a system status code into an error message of the format:

```
%    FACILITY–L–CODE, TEXT
–
```

where:

% or –	Denotes the first message (%) or a subsequent message (–) from a single error.
FACILITY	Is the name of the VAX/VMS component that generated the error.
L	Is the severity level. It can be S, I, W, E, or F. (See Section 5.1.2.)

CODE Is an abbreviation of the message text. Message descriptions in the documentation for that facility are normally alphabetized by this code.

TEXT Explains the message.

This program expands the previous example by invoking the SYS$GETMSG system service and passing it the status code returned from the SYS$TRNLOG system service.

```
100     EXTERNAL INTEGER FUNCTION       SYS$TRNLOG,     &
                                        SYS$GETMSG
        EXTERNAL INTEGER CONSTANT       SS$_NORMAL,     &
                                        SS$_NOTRAN,     &
                                        SS$_ACCVIO,     &
                                        SS$_IVLOGNAM,   &
                                        SS$_RESULTOVF,  &
                                        SS$_BUFFEROVF,  &
                                        SS$_MSGNOTFND

        DECLARE LONG SYS_STATUS, SAVE_STATUS, WORD RSN_LEN, STRING L_NAME

        COMMON (MSGCOM) STRING RSN_BUF = 80, MESSAGE = 80

        LINPUT "Logical name for translation"; L_NAME

200 SYS_STATUS = SYS$TRNLOG(L_NAME, RSN_LEN, RSN_BUF,,,,)

        SELECT SYS_STATUS

        CASE SS$_NORMAL
                PRINT "Equivalence name is: "; SEG$(RSN_BUF,1,RSN_LEN)

        CASE SS$_NOTRAN
                PRINT "No translation for logical name"

        CASE  SS$_ACCVIO
                PRINT "Error when trying to read or write"
                PRINT "the logical name you provided."

        CASE SS$_IVLOGNAM
                PRINT "Invalid logical name"

        CASE SS$_RESULTOVF
                PRINT "Returned string too large for"
                PRINT "receiving buffer."

        CASE ELSE
                PRINT "Unknown error from"
                PRINT "SYS$TRNLOG, error is"; SYS_STATUS
        END SELECT


        SAVE_STATUS = SYS_STATUS
        SYS_STATUS = SYS$GETMSG(SAVE_STATUS BY VALUE,   &
                                RSN_LEN,                &
                                MESSAGE,,,)

        PRINT "Message is: "; SEG$(MESSAGE, 1, RSN_LEN)

        SELECT SYS_STATUS
        CASE SS$_NORMAL
                PRINT "Successful translation of status code"
```

```
CASE SS$_BUFFEROVF
        PRINT "Returned string too large for"
        PRINT "receiving buffer."
CASE SS$_MSGNOTFND
        PRINT "Translation not successful."
CASE ELSE
        PRINT "Unknown error from"
        PRINT "SYS$GETMSG, error is"; SYS_STATUS
END SELECT

END
```

Line 100 starts with the necessary external and internal declarations for both SYS$TRNLOG and SYS$GETMSG. It ends with a LINPUT statement prompting for a logical name to translate. Thus, it declares the SYS$TRNLOG and SYS$GETMSG routines to be EXTERNAL FUNCTIONs returning longword status codes, then goes on to declare SS$_NORMAL, SS$_NOTRAN, SS$_ACCVIO, SS$_IVLOGNAM, SS$_RESULTOVF, SS$_BUFFEROVF, and SS$_MSGNOTFND to be longword EXTERNAL CONSTANTs. The values of these symbolic constants are the possible return values of the system services.

The line then declares local storage for the SYS_STATUS variable that receives the returned status code, the SAVE_STATUS variable that is used to pass the code returned by SYS$TRNLOG to SYS$GETMSG, a WORD variable called RSN_LEN that receives the length of the returned strings, and a STRING variable called L_NAME. The COMMON statement declares two 80–character fixed-length string variables to receive the returned equivalence name and the returned message translation.

Line 200 invokes the SYS$TRNLOG service and tests the result, as described in the previous section. The program then copies the returned status code into the SAVE_STATUS variable and passes this as a parameter to the SYS$GETMSG service.

SYS$GETMSG requires five parameters:

- A status code value. Note that SYS$GETMSG requires this parameter to be passed BY VALUE. You must specify BY VALUE; otherwise BASIC uses the default parameter passing mechanism (BY REF).
- The address of a WORD variable to receive the returned string's length. Because the default parameter passing mechanism is BY REF, you specify only the variable, RSN_LEN.
- The address of the string descriptor pointing to the returned string. Because the default parameter passing mechanism for strings is BY DESC, you specify only a string variable, MESSAGE.
- A 4–bit mask determining which parts of the message SYS$GETMSG returns. If you want to specify a mask, you must override the default parameter passing mechanism and specify BY VALUE. However, this program accepts the default, which is to receive all parts of the message.
- The address of a 4–byte array for other information. This example passes a null parameter, accepting the default.

The program then displays the returned message using the SEG$ function. The SEG$ function lets you examine only as many characters as the system service assigned to the MESSAGE variable.

The program then uses the SELECT statement to test the value of the code returned in SYS_STATUS. Each CASE block specifies one of the possible symbolic return values and the actions to be taken based on this value. The CASE ELSE block handles any unexpected errors.

5.1.4.4 Queueing I/O Requests: SYS$QIOW

SYS$QIOW initiates an input or output request to a channel. After queueing the request, the program waits for the request to complete. SYS$QIOW uses VAX/VMS named constants called function codes to specify the type of operation performed. See the *VAX/VMS System Services Reference Manual* and the *VAX/VMS I/O User's Guide* for more information on function codes. This example calls SYS$QIOW to write text to the controlling terminal.

```
10      EXTERNAL INTEGER FUNCTION       SYS$ASSIGN,      &
                                        SYS$TRNLOG,      &
                                        SYS$QIOW
        EXTERNAL INTEGER CONSTANT       IO$_WRITEVBLK,   &
                                        SS$_NORMAL,      &
                                        SS$_NOTRAN

        COMMON STRING BUF_FER = 14

        MAP (EQNAM) STRING RSN_BUF   = 80
        MAP (EQNAM) STRING FILL = 4, RSN_NAM = 76

        MAP (IOSB) LONG IO_SB(1)
        MAP (IOSB) WORD IO_SB_W(3)

        DECLARE LONG SYS_STATUS, WORD CHAN, RSN_LEN, STRING DEV_NAM

 TRANSLATE_ROUTINE:

        SYS_STATUS = SYS$TRNLOG("SYS$OUTPUT", RSN_LEN, RSN_BUF,,,,)
        SELECT SYS_STATUS
                CASE SS$_NORMAL
                CASE SS$_NOTRAN
                CASE ELSE
                        PRINT "Error from"
                        PRINT "SYS$TRNLOG, error is"; SYS_STATUS
                        GOTO DONE
        END SELECT

        IF ASCII(RSN_BUF) = ASCII (ESC)
        THEN DEV_NAM = SEG$(RSN_NAM, 4, RSN_LEN)
        ELSE DEV_NAM = SEG$(RSN_BUF, 1, RSN_LEN)
        END IF


 ASSIGN_ROUTINE:

        SYS_STATUS = SYS$ASSIGN(DEV_NAM, CHAN,,,)

        IF (SYS_STATUS AND 1%) <> 1
        THEN
                PRINT "Error from SYS$ASSIGN"
                PRINT "Error number is"; SYS_STATUS
                GOTO DONE
        END IF

 WRITE_ROUTINE:

        BUF_FER = "This is a test"

        SYS_STATUS = SYS$QIOW(, CHAN BY VALUE,                 &
                              IO$_WRITEVBLK BY VALUE,          &
                              IO_SB() BY REF,,,                &
                              BUF_FER BY REF,                  &
                              14% BY VALUE,,                   &
                              32% BY VALUE,,)
```

```
        IF (SYS_STATUS AND 1%) = 0%
        THEN PRINT "Error from SYS$QIOW"
             PRINT "Error is";SYS_STATUS
             GOTO DONE
        END IF


        PRINT "Contents of I/O Status Block"
        PRINT "IO_SB(0) = ";IO_SB(0)
        PRINT "IO_SB(1) = ";IO_SB(1)


DONE:
        END
```

Line 10 defines SYS$ASSIGN, SYS$TRNLOG, and SYS$QIOW as EXTERNAL FUNCTIONs, and declares the global symbol IO$_WRITEVBLK to be an external constant. IO$_WRITEVBLK is a function code that specifies to SYS$QIOW the I/O function to be performed. IO$_WRITEVBLK means "write virtual block." The program then defines a COMMON to hold a fixed-length string variable named BUF_FER. This variable holds the data to be written to the terminal.

The MAP named EQNAM receives the equivalence name from SYS$TRNLOG. This buffer is mapped in two ways because the returned equivalence name may refer to a process-permanent file. If it does, the first four characters contain VAX/VMS control information and must be excluded from the device specification. The MAP named IOSB holds the I/O status block array that holds information returned by the SYS$QIOW service.

The program then declares:

- The SYS_STATUS variable that receives the return status from all system service calls
- The CHAN variable that receives the channel number from the SYS$ASSIGN service
- The RSN_LEN variable that holds the length of the string returned by the SYS$ASSIGN service
- The DEV_NAM string variable that is used to pass a device name to the SYS$QIOW service
- The QIO_CHAN variable that is used to pass the channel number to the SYS$QIOW service

The block labeled TRANSLATE_ROUTINE: calls SYS$TRNLOG to find the equivalence name of SYS$OUTPUT. The resulting equivalence name string is assigned to RSN_BUF, and its length is assigned to RSN_LEN. It then tests the returned status code for the result of the SYS$TRNLOG call. Note that the SELECT statement takes no action if the call returns SS$_NORMAL or SS$_NOTRANS. Thus these two status codes cause control to fall through to the statement following the END SELECT statement, while any other status codes cause the program to end.

The program then tests the equivalence name string to see if SYS$OUTPUT is a terminal or a process-permanent file. If the first character is an escape, SYS$OUTPUT is a process-permanent file, and DEV_NAM is assigned the value in RSN_NAM, which excludes the first three characters of RSN_BUF. If the first character is not an escape, DEV_NAM is assigned the value in RSN_BUF.

The block labeled ASSIGN_ROUTINE: uses SYS$ASSIGN to assign the device specified by DEV_NAM. The routine then tests the return value, ending the program if the call was not successful.

The block labeled WRITE_ROUTINE: assigns test text to the variable BUF_FER. The routine then calls SYS$QIOW to queue an I/O request.

This type of I/O request is specified by IO$_WRITEVBLK and the output device is the controlling terminal. SYS$QIOW accepts up to six parameters, plus six optional device- and function-specific I/O request parameters:

- The number of an event flag to be set when the I/O request is complete. The example accepts the default (event flag 0).
- The I/O channel number of the device to which the request is directed. This example uses the CHAN variable, assigned a value by SYS$ASSIGN.
- The symbolic function code and modifier bits specifying the operation to be performed. The example uses IO$_WRITEVBLK.
- The address of a quadword I/O status block to receive final completion status. In the example, this parameter is explicitly passed BY REF, because the default parameter passing mechanism for arrays is BY DESC.
- The address of the entry mask for an asynchronous service routine to be executed when the I/O completes. The example passes a null parameter, specifying that there is no such service routine.
- Additional parameters for the asynchronous service routine. The example passes a null parameter, specifying none.
- Optional device- and function-specific parameters. The example specifies a 14–byte buffer and a carriage control parameter. The other parameters are null.

The program then tests the return value of the SYS$QIOW call and ends the program if the call was not successful. If the call was successful, the program prints the values in the I/O status block. Note that the value in IO_SB(0) is a longword integer. The high-order word contains the number of bytes transferred; the low-order word contains the operation status.

The block labeled DONE: ends the program.

5.1.4.5 Getting Information About a Job/Process: SYS$GETJPI

The Get Job/Process Information system service provides accounting, status, and identification information about a specified process. The following example uses SYS$GETJPI to return the user name associated with the process running this program. This program uses the RECORD statement to create the data structure filled in by SYS$GETJPI.

```
50      EXTERNAL INTEGER FUNCTION SYS$GETJPI
        EXTERNAL INTEGER CONSTANT JPI$_PRCNAM
        EXTERNAL INTEGER CONSTANT JPI$_ACCOUNT
        EXTERNAL INTEGER CONSTANT JPI$_CPUTIM
        EXTERNAL INTEGER CONSTANT SS$_NORMAL

        RECORD JPIBUF
                WORD BUFFER_LENGTH1
                WORD ITEM_CODE1
                LONG BUFFER_ADDRESS1
                LONG RETURN_LENGTH_ADDRESS1
                WORD BUFFER_LENGTH2
                WORD ITEM_CODE2
                LONG BUFFER_ADDRESS2
                LONG RETURN_LENGTH_ADDRESS2
                WORD BUFFER_LENGTH3
                WORD ITEM_CODE3
                LONG BUFFER_ADDRESS3
                LONG RETURN_LENGTH_ADDRESS3
                LONG LIST_TERMINATOR
        END RECORD JPIBUF
```

```
        DECLARE JPIBUF ITEMS

        MAP (BUFFER_AREA1) STRING USER_NAME = 15
        MAP (BUFFER_AREA2) STRING ACCOUNT_NAME = 8
        MAP (BUFFER_AREA3) LONG CPU_TIME

        DECLARE WORD USER_NAME_LENGTH
        DECLARE WORD ACCOUNT_LENGTH
        DECLARE WORD CPU_TIME_LENGTH
        DECLARE LONG SYS_STATUS

LOAD_RECORD:

        ITEMS::BUFFER_LENGTH1 = 15
        ITEMS::ITEM_CODE1      = JPI$_PRCNAM
        ITEMS::BUFFER_ADDRESS1 = LOC(USER_NAME)
        ITEMS::RETURN_LENGTH_ADDRESS1 = LOC(USER_NAME_LENGTH)
        ITEMS::BUFFER_LENGTH2 = 8
        ITEMS::ITEM_CODE2        = JPI$_ACCOUNT
        ITEMS::BUFFER_ADDRESS2 = LOC(ACCOUNT_NAME)
        ITEMS::RETURN_LENGTH_ADDRESS2 = LOC(ACCOUNT_LENGTH)
        ITEMS::BUFFER_LENGTH3 = 4
        ITEMS::ITEM_CODE3 = JPI$_CPUTIM
        ITEMS::BUFFER_ADDRESS3 = LOC (CPU_TIME)
        ITEMS::RETURN_LENGTH_ADDRESS3 = LOC (CPU_TIME_LENGTH)
        ITEMS::LIST_TERMINATOR = 0

CALL_JPI:

        SYS_STATUS = SYS$GETJPI(,,, ITEMS ,,,)

        IF SYS_STATUS <> SS$_NORMAL
        THEN
                PRINT "Error from SYS$GETJPI"
                PRINT "Error number is";SYS_STATUS
                GOTO DONE
        END IF


        PRINT "Username is: "; USER_NAME
        PRINT "Account name is: "; ACCOUNT_NAME
        PRINT "CPU time in 10-millisecond ticks is: "; CPU_TIME

DONE:
        END
```

Line 50 declares SYS$GETJPI as an EXTERNAL FUNCTION and JPI$_PRCNAM, JPI$_ACCOUNT, JPI$_CPUTIM and SS$_NORMAL as external constants. The program then uses the RECORD statement to create a template for the descriptor block required by SYS$GETJPI. This template is named JPIBUF. This descriptor block requires 12 bytes for each item code used by SYS$GETJPI. The first word contains the length, in bytes, of the buffer to which information is returned. The second word contains the item code specifying the type of information returned. The next longword contains the address of the buffer. The last longword contains the address of a WORD integer to receive the length of the returned information.

This is repeated twice more because this SYS$GETJPI invocation asks for a total of three pieces of information. As described in the documentation for SYS$GETJPI, the item descriptor is terminated by a longword containing a value of zero.

The program then declares an instance of this template by declaring ITEMS to be of type JPIBUF.

The next three MAP statements allocate storage for the information returned by SYS$GETJPI; a 15–byte string for the username, an 8–byte string for the account name, and a longword (4 bytes) for the CPU time.

The next four DECLARE statements declare the storage for: 1) the word-length integers that receive the length of the information returned, and 2) the longword that receives the status code from SYS$GETJPI.

The block labeled LOAD_RECORD: assigns values to the item descriptor block. The buffer length for the process name is 15 bytes. The item code is assigned the value of the symbolic constant JPI$_PRCNAM. The addresses of the buffer and the word integer receiving the length of the returned information are assigned with the LOC function. Similarly, the values for the two other SYS$GETJPI parameters are assigned. Finally, the descriptor block is terminated by assigning a value of zero to LIST_TERMINATOR.

The block labeled CALL_JPI: invokes SYS$GETJPI as a function. SYS$GETJPI assigns the status code to SYS_STATUS. Note that the only parameter specified is ITEMS, the RECORD variable of data-type JPIBUF. BASIC passes this parameter BY REF, thus specifying the start of the entire descriptor block.

The parameter list accepts defaults for event flags, process identification, process name, I/O status block, AST address, and AST parameters. Because this system service call does not specify a separate process, information is returned about the process issuing the call.

The program then tests the value returned in SYS_STATUS and ends the program if the code is anything other than SS$_NORMAL. If the return value is SS$_NORMAL, the program displays the returned information at the terminal.

5.1.5 Resolving External Names

In the previous system service examples, the linker fills in the values for all symbolic names (for example, SS$_NORMAL) by resolving the symbol in the default system library. However, there are some symbolic names that are defined only in the system Macro library, STARLET.MLB.

To resolve external names that are defined only in STARLET.MLB, you must write a 2–line VAX–11 MACRO program invoking the required symbol definition macro, assemble the VAX–11 MACRO program, and link the resulting object module with the BASIC program that uses these symbolic names. For example, to use the $LKWSET system service, you must code this VAX–11 MACRO program to invoke the $LKWSETDEF macro:

```
$LKWSETDEF <GLOBAL>
.END
```

This program defines all the symbolic constants used by the $LKWSET system service.

5.2 Calling Run-Time Library Routines

The Run-Time Library is a collection of procedures that you can call from a BASIC program. The Run-Time Library contains over 600 procedures that let you:

- Perform a wide variety of mathematical functions. This is especially useful for languages that do not have those functions built in.
- Write programs that use the cursor control features of the video terminal, without needing to know what type of terminal is being used.

- Allocate resources for your process, such as virtual memory.
- Convert one data type to another and format the result for output.
- Obtain the system date and time in a wide variety of formats.
- Gain access to machine instructions and system services that are difficult to use in high-level language programs.

If a BASIC language element performs the same function as an RTL routine, you should use the BASIC language element.

RTL procedures may require that a parameter be passed in a certain way. See Chapter 3 for a discussion of parameter passing mechanisms.

Like the system services library, the RTL also uses symbolic constants. Most of the routines in the RTL are located in the RTL shareable image, VMSRTL.EXE. Others are found in the default system object module library, STARLET.OLB. The linker automatically searches these files to resolve symbols. See Chapter 10 for more information about these system libraries and executable images.

5.2.1 Types of RTL Procedures

RTL procedures behave in one of three ways:

- Some RTL procedures behave like system services in that they return:
  - A status code describing the success or failure of the operation
  - Information by way of output parameters specified in the parameter list

  RTL procedures of this type should be accessed as external functions; the value the function returns is the status code.
- Other RTL procedures behave like BASIC functions; they return a function value instead of a status code. Should an error occur in a procedure of this type, the error is signaled to the calling program. If the RTL error corresponds to a BASIC error (for example, division by zero), you can trap the error in a BASIC error handler. Procedures of this type should also be accessed as external functions.
- Finally, there are RTL procedures that behave like SUB subprograms; they do not return any information as a return value, and they signal errors. Procedures of this type must be accessed with the CALL statement; they cannot be accessed as external functions. In these procedures, information is written to an output parameter specified in the CALL statement. Only procedures that return HFLOAT values behave this way.

To find out which way an RTL procedure behaves, see the *VAX–11 Run-Time Library Reference Manual*. The body of the manual contains a complete description of each procedure, including a functional specification, a calling sequence, a list of arguments, and a list of possible error messages. Appendix A of the manual contains abbreviated procedure descriptions, including the calling sequence and a special notation that lists the characteristics of each argument. If a procedure's format begins with:

- Ret-status, the procedure behaves like a system service; it returns a status code describing the success or failure of the operation, and returns other information by way of output parameters.
- A value, the procedure behaves like a BASIC function; it returns only the function value, and signals errors.

- CALL, the procedure behaves like a SUB subprogram; it does not have a return value. Instead, it returns information by way of output parameters.

The following sections provide examples of invoking these types of RTL procedures.

5.2.2 Measuring Performance: LIB$INIT_TIMER, LIB$FREE_TIMER, and LIB$STAT_TIMER

The routines in the RTL performance package let you measure your program's elapsed time, CPU time, buffered I/O, direct I/O, and page faults. This example uses the RTL timer routines and the system service SYS$ASCTIM:

```
100     EXTERNAL INTEGER FUNCTION LIB$INIT_TIMER
        EXTERNAL INTEGER FUNCTION LIB$STAT_TIMER
        EXTERNAL INTEGER FUNCTION LIB$FREE_TIMER
        EXTERNAL INTEGER CONSTANT SS$_NORMAL

        DECLARE LONG COND_VALUE, CODE, HANDLE
        DECLARE STRING TIME_BUFFER
        HANDLE = 0
        TIME_BUFFER = SPACE$(50%)

        MAP (TIMER) LONG ELAPSED_TIME, FILL
        MAP (TIMER) LONG CPU_TIME
        MAP (TIMER) LONG BUFIO
        MAP (TIMER) LONG DIRIO
        MAP (TIMER) LONG PAGE_FAULTS

 WHILE -1%

 INPUT_ROUTINE:

        PRINT "This program returns information about:"
        PRINT "Elapsed time (1)"
        PRINT "CPU time (2)"
        PRINT "Buffered I/O (3)"
        PRINT "Direct I/O (4)"
        PRINT "Page faults (5)"
        PRINT "Enter zero to exit program"
        PRINT "Enter a number from one to"
        PRINT "five for performance information"
        INPUT "One, two, three, four, or five"; CODE
        PRINT

        GOTO DONE IF CODE = 0

 INIT_ROUTINE:

        COND_VALUE = LIB$INIT_TIMER( HANDLE )

        IF (COND_VALUE <> SS$_NORMAL)
        THEN
                PRINT "Error in initialization"
                GOTO DONE
        END IF


 WASTE_TIME:

        A = 0                   !
        FOR I = 1 to 100000     !
          A = A + 1             !This code merely uses some CPU time
        NEXT I          !
```

```
MEASURE_TIME:

        COND_VALUE = LIB$STAT_TIMER( CODE, ELAPSED_TIME, HANDLE )

        IF (COND_VALUE <> SS$_NORMAL)
        THEN
                PRINT "Error in statistics routine"
                GOTO DONE
        END IF

        GOTO PRINT_ROUTINE IF CODE <> 1%
        CALL SYS$ASCTIM ( , TIME_BUFFER, ELAPSED_TIME, 1% BY VALUE)
        PRINT "Elapsed time: "; TIME_BUFFER


PRINT_ROUTINE:

        PRINT "CPU time in seconds: "; .01 * CPU_TIME IF CODE = 2%
        PRINT "Buffered I/O: ";BUFIO IF CODE = 3%
        PRINT "Direct I/O: ";DIRIO IF CODE = 4%
        PRINT "Page faults: ";PAGE_FAULTS IF CODE = 5%
        PRINT "Return value: "; COND_VALUE
        PRINT

NEXT

DONE:
        COND_VALUE = LIB$FREE_TIMER( HANDLE )
        IF (COND_VALUE <> SS$_NORMAL)
        THEN
                PRINT "Error in LIB$FREE_TIMER"
        END IF

        END
```

Line 100 defines LIB$INIT_TIMER, LIB$STAT_TIMER, and LIB$FREE_TIMER as EXTERNAL FUNCTIONs and SS$_NORMAL as an EXTERNAL CONSTANT. The program then declares COND_VALUE (the return status from the RTL procedures), CODE (the parameter that describes the type of information requested from LIB$STAT_TIMER), and HANDLE (the parameter receiving the storage address used by the timer routines to return information) as longword integers.

The next four statements declare the string TIME_BUFFER (the location used by SYS$ASCTIM to store the returned string), pre-extend this string with the SPACE$ function, and initialize HANDLE to zero.

The MAP statements allocate storage for the returned information. The storage for these variables is overlaid because LIB$STAT_TIMER uses the same storage regardless of the type of information specified with the CODE parameter.

The next five labeled blocks are inside a WHILE loop.

The block labeled INPUT_ROUTINE: prompts for the type of information needed. CODE is assigned a value between zero and five. If the value of CODE is zero, control is transferred to the DONE: label.

The block labeled INIT_ROUTINE: calls the timer initialization routine, LIB$INIT_TIMER. The value of zero in the HANDLE parameter specifies that LIB$INIT_TIMER is to allocate storage for the return information and return the virtual address used in the HANDLE parameter. This value is an input parameter to the other timer routines.

The program compares the return status (COND_VALUE) with SS$_NORMAL. If the return status does not equal SS$_NORMAL, the program prints an error message and exits.

The block labeled WASTE_TIME: does not do anything useful; it merely uses some CPU time.

The block labeled MEASURE_TIME: calls the timer statistics routine, LIB$STAT_TIMER. The CODE parameter specifies the type of information to be returned. The ELAPSED_TIME parameter specifies the location in which to store the returned information (the MAP named TIMER).

The HANDLE parameter contains the address returned by the LIB$INIT_TIMER routine. Each invocation of LIB$STAT_TIMER compares current statistics with the information in the block pointed to by HANDLE, then updates this HANDLE block. Thus, the information returned in the storage specified by the TIMER MAP is incremental. The program compares the return status (COND_VALUE) with SS$_NORMAL. If the return status does not equal SS$_NORMAL, the program prints an error message and exits.

The program transfers control to the block labeled PRINT_ROUTINE: if CODE has a value other than one. If CODE equals one, then LIB$STAT_TIMER returns an elapsed time value as a 64–bit binary value. If this is the case, the program calls the system service SYS$ASCTIM to translate this 64–bit binary value to an ASCII string. Note that SYS$ASCTIM is accessed with the CALL statement. This means that SYS$ASCTIM need not be declared as an external function and that no status code is returned.

The block labeled PRINT_ROUTINE: checks the value of CODE and prints the appropriate header depending on this value. LIB$STAT_TIMER returns CPU time in 10–millisecond ticks; therefore this value is multiplied by .01 to show CPU time in seconds.

The block labeled DONE: calls LIB$FREE_TIMER. This routine frees the storage allocated by LIB$INIT_TIMER. It then compares the return status (COND_VALUE) with SS$_NORMAL. If the return status does not equal SS$_NORMAL, the program prints an error message and exits.

5.2.3 Using Logical Unit Numbers: LIB$GET_LUN and LIB$FREE_LUN

Logical unit numbers (LUNs) can be used as channel numbers for BASIC I/O operations. Using LIB$GET_LUN to find a free channel number allows for better program modularity. For example, if you write a subprogram that opens a file, you must be certain that the channel number specified in the subprogram is not being used for I/O in the calling program. If the channel number is already in use, an OPEN statement in a subprogram closes any file open on that channel and opens the new file. Using a channel number returned by LIB$GET_LUN ensures that you open the file on an unused channel.

LIB$GET_LUN returns a number between 100 and 119, inclusive. Channel numbers in this range are reserved for programs that use LIB$GET_LUN and LIB$FREE_LUN. If you specify a channel number as a constant, it must be in the range 0 to 99, inclusive.

This example shows a subprogram allocating two LUNs and using them as channel expressions when copying one file to another. After processing is finished, the LUNs are returned to the system resource pool with the LIB$FREE_LUN routine:

```
100     SUB GETLUN ( STRING INFILE, OUTFILE )

200     ON ERROR GOTO 19000

300     EXTERNAL INTEGER FUNCTION LIB$GET_LUN
        EXTERNAL INTEGER FUNCTION LIB$FREE_LUN
        EXTERNAL INTEGER CONSTANT SS$_NORMAL
        EXTERNAL INTEGER CONSTANT LIB$_INSLUN
```

```
400     DECLARE LONG    COND_VALUE,          &
                        FIRST_CHAN,          &
                        SECOND_CHAN

500     MAP (XYZ) STRING ABC = 132

600     COND_VALUE = LIB$GET_LUN( FIRST_CHAN )

700     IF COND_VALUE = LIB$_INSLUN THEN PRINT "No free LUNs"
                                          SUBEXIT

800     COND_VALUE = LIB$GET_LUN( SECOND_CHAN )

900     IF COND_VALUE = LIB$_INSLUN THEN PRINT "No free LUNs"
                                          SUBEXIT

1000    OPEN INFILE FOR INPUT AS FILE #FIRST_CHAN,  &
                        SEQUENTIAL VARIABLE,    &
                        MAP XYZ

1100    OPEN OUTFILE FOR OUTPUT AS FILE #SECOND_CHAN,  &
                        SEQUENTIAL VARIABLE,    &
                        MAP XYZ

1200    WHILE -1%
                GET #FIRST_CHAN
                PUT #SECOND_CHAN, COUNT RECOUNT
        NEXT

19000   IF (ERR = 11) AND (ERL = 1200) THEN
                        PRINT "File has been copied"
                        RESUME 32700
                       ELSE ON ERROR GO BACK

32700   CLOSE #FIRST_CHAN, SECOND_CHAN
        COND_VALUE = LIB$FREE_LUN( FIRST_CHAN )
        COND_VALUE = LIB$FREE_LUN( SECOND_CHAN )

32767   SUBEND
```

- Line 100 identifies the module as a subprogram with two input parameters, an input file and an output file.
- Line 200 transfers control to line 19000 if an error occurs.
- Line 300 defines LIB$GET_LUN and LIB$FREE_LUN as external functions and SS$_NORMAL and LIB$_INSLUN as external constants.
- Line 400 declares storage for COND_VALUE (the condition code returned by LIB$GET_LUN and LIB$FREE_LUN), FIRST_CHAN (the variable to receive the first LUN), and SECOND_CHAN (the variable to receive the second LUN).
- Line 500 allocates a single record buffer for both the input and output files.

- Line 600 invokes LIB$GET_LUN. This routine returns a currently unused LUN in the FIRST_CHAN parameter. The routine's status is returned in COND_VALUE.
- Line 700 tests the value of COND_VALUE. If the return status is equal to the symbolic constant LIB$_INSLUN, the operation failed because there were no LUNs available. If this is the case, the program prints a message and transfers control to the SUBEND statement.
- Line 800 invokes LIB$GET_LUN for the second time, specifying that the second LUN be returned in the SECOND_CHAN parameter.
- Line 900 tests the value of COND_VALUE, as in line 700.
- Line 1000 opens the input file using the LUN returned in FIRST_CHAN.
- Line 1100 opens the output file using the LUN returned in SECOND_CHAN.
- Line 1200 starts a WHILE loop that performs sequential GETs from the input file and sequential PUTs to the output file. Because the two files share MAP (XYZ), this loop copies records from the input file to the output file. Because the PUT statement specifies COUNT RECOUNT, each PUT writes out the same number of bytes as were read in the previous GET.
- Line 19000 contains the error handler. If the error was end-of-file and occurred on line 1200, the handler prints a message and transfers control to line 32700. If any other error occurs, control is transferred to the calling program.
- Line 32700 closes all files, then invokes LIB$FREE_LUN twice to release the LUNs to the system pool.
- Line 32767 ends the subprogram.

5.2.4 Using Arccosine Procedures: MTH$ACOS

The MTH$ACOS procedure returns the angle (in radians) whose cosine you supply as an input parameter. MTH$ACOS does not return a condition code describing the success or failure of the operation. If an error occurs, the procedure resignals the error to the calling program. However, the only possible error from the MTH$ACOS routine is "Invalid argument" — an error for which there is no corresponding BASIC error. A BASIC error handler cannot trap this error; therefore, the calling program checks the input value to make sure it falls in the allowable range before invoking the routine. For example:

```
100     EXTERNAL SINGLE FUNCTION MTH$ACOS
        DECLARE SINGLE COS_VALUE, ANGLE
 INPUTROUTINE:
        INPUT "Cosine value between -1 and +1"; COS_VALUE
        IF (COS_VALUE < -1) OR (COS_VALUE > 1)
        THEN
                PRINT "Invalid cosine value"
                GOTO INPUTROUTINE
        END IF
500     ANGLE = MTH$ACOS( COS_VALUE )
        PRINT "The angle with that cosine is ";ANGLE;"radians"
        END
```

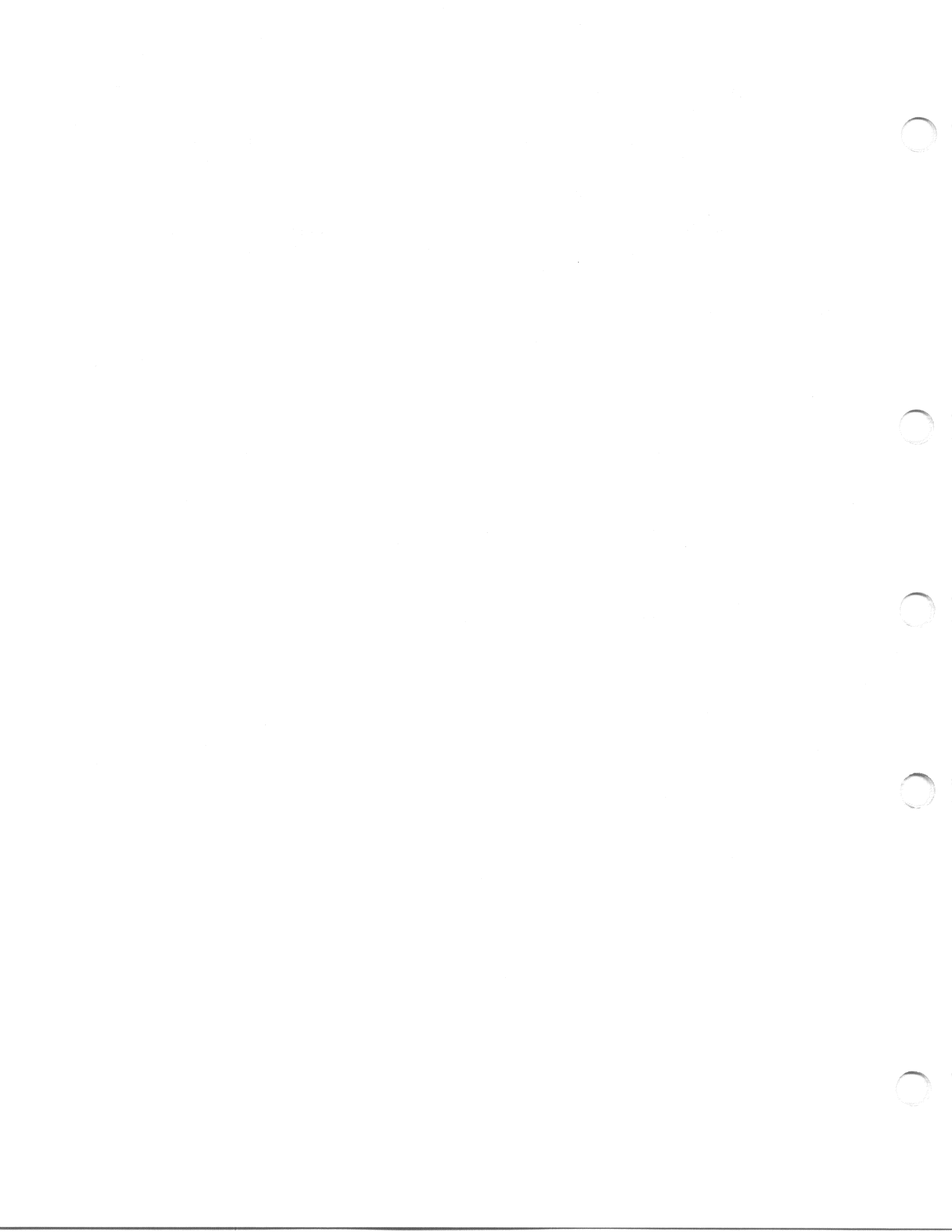
Line 100 defines MTH$ACOS as an EXTERNAL FUNCTION, returning a SINGLE value. It then declares storage for two SINGLE variables: COS_VALUE and ANGLE.

The block labeled INPUT_ROUTINE prompts for an input cosine value and checks it to make sure it is in the allowed range.

Line 500 invokes MTH$ACOS. The input parameter is the cosine value, and the routine returns an angle into variable ANGLE. The program then prints the returned information and ends.

Note

Be sure that the data-type keyword in the EXTERNAL statement matches the data type returned by the RTL procedure. If you specify an incorrect data type, your program misinterprets the returned value and does not signal an error.



Chapter 6
The RECORD Statement

This chapter explains how to use the RECORD statement.

The RECORD statement provides a way to create data structures in BASIC. You use the RECORD statement to declare the form or *template* of a data structure. BASIC treats this structure as a user-defined data type. Once this user-defined data type has been declared, it can be used to allocate instances of the template.

This provides you with a mechanism for accessing a collection of variables using a single name. This capability is useful in defining I/O buffers and passing information to routines. In addition, when you extract record definitions from the VAX–11 Common Data Dictionary, BASIC translates the CDD definition to a RECORD statement. See Chapter 9 for more information about the CDD.

6.1 The 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 the RECORD name anywhere a BASIC data-type keyword is valid. Thus, you build the data structure using: 1) variables of any valid BASIC data type or 2) previously defined RECORD data types. The format of the RECORD statement is:

```
RECORD record-name
        {data-type component-name,...} ,...
              .
              .
              .
END RECORD [record-name]
```

where:

record-name	Is the name of the data structure. This name must conform to the rules for naming BASIC variables.
data-type	Is any valid BASIC data-type keyword or another record-name.
component-name	Is a variable, array or FILL item. In a record definition, variables and arrays of standard BASIC data types are called elementary record components.

The declarations between the RECORD statement and the END RECORD statement are called a RECORD block. Each line of text in a RECORD block can have an optional line number.

If the data type of a RECORD element is STRING, the string is fixed-length and you can supply an optional string length. If you do not specify a string length, the default is 16. For example:

```
100     RECORD EMPLOYEE
110         LONG EMP_NUMBER
120         STRING FIRST_NAME = 10
130         STRING LAST_NAME = 20
140     END RECORD EMPLOYEE
```

The RECORD statement does not allocate any storage; it only defines and names a data structure. Thus the record declaration is called a *record template*. Once the structure has been defined, you can use the record template as a data type in a statement such as DECLARE, MAP, or COMMON. Using the record template this way is called declaring a *record instance*. It is the declaration of the record instance that actually allocates the storage for the RECORD.

To continue the previous example:

```
1000    DECLARE EMPLOYEE EMP_REC
```

This statement declares a variable named EMP_REC. EMP_REC is of data-type EMPLOYEE and requires 34 bytes of storage: a longword for the LONG integer component EMP_NUM, 10 characters for the string component FIRST_NAME, and 20 characters for the string component LAST_NAME.

Whenever you access an elementary record component, that is, a variable named in a RECORD definition, you do it in the context of the record instance. Therefore, RECORD component names need not be unique in your program. For example, you can have a record component named FIRST_NAME in any number of different RECORD definitions. References to this component are unambiguous because every record component reference must specify the record instance in which it resides. However, you still cannot use a BASIC keyword as a RECORD component name.

To access a particular elementary component within a record, you use the name of the declared variable and the elementary component name, separated by two colons (::). For example:

```
2000    LINPUT "Employee number";EMP_REC::EMP_NUMBER
2010    LINPUT "First name";EMP_REC::FIRST.NAME
2020    LINPUT "Last name";EMP_REC::LAST.NAME
```

Line 1000 in the previous example declares EMP_REC to be an instance of the record template EMPLOYEE. Lines 2000, 2010, and 2020 all reference the components of this record instance.

6.1.1 Grouping RECORD Components

A RECORD component may also be a named group of variables, identified with the keyword GROUP. The GROUP name can be followed by a list of bounds, thus defining an array of the GROUP components. GROUP is valid only within a RECORD block and has the format:

```
GROUP group-nam[(b1,b2...b32)]
      {data-type group-component,...} ,...
          .
          .
          .
END GROUP [group-nam]
```

where:

group-nam	Is the name of block of record components.
b1,b2...b32	Are the bounds of the group array. This array is zero-based, that is, BASIC always allocates the zeroth element of this array.
data-type	Is any valid BASIC data-type keyword or another record-name.
group-component	Is a variable, array or FILL item.

Each line of text in a GROUP block can have an optional line number. The declarations between the GROUP statement and the END GROUP statement are called a GROUP block. For example:

```
1000    RECORD YACHT

                GROUP TYPE_OF_YACHT
                        STRING MANUFACTURER = 10
                        STRING MODEL = 10
                END GROUP TYPE_OF_YACHT

                GROUP SPECIFICATIONS
                        STRING RIG = 6, LENGTH_OVER_ALL = 3
                        DECIMAL(5,0) DISPLACEMENT
                        DECIMAL(2,0) BEAM
                        DECIMAL(7,2) PRICE
                END GROUP SPECIFICATIONS

        END RECORD YACHT
```

This example declares the record 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.

Note that BASIC also allows GROUPs within GROUPs.

Once you have defined the YACHT data type, you can declare instances of this data type:

```
2000    DIM YACHT FLEET(100)
```

This statement allocates a 101–element array of data-type YACHT.

To access a particular elementary component within this record, you can 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 (::).

The name of the declared record instance, or RECORD name, is always required when referencing a RECORD component.

Thus, the record name *qualifies* the group name and the group name qualifies the elementary record component. The elementary component name, qualified by all intermediate group names, qualified by the record name, is called a *fully qualified* component. The full qualification of a component is also called a *component path name*.

GROUP names are optional unless: 1) there is more than one record component with the same name, or 2) the GROUP is an array. For example:

```
3000    FOR I% = 1% TO 100%
3010        INPUT "Manufacturer"; FLEET(I%)::MANUFACTURER
3020        INPUT "Model"; FLEET(I%)::TYPE_OF_YACHT::MODEL
3030    NEXT I%
```

In this case, specifying group TYPE_OF_YACHT in line 3020 is valid but not required.

The GROUP block provides a way to impose a logical structure in a RECORD definition, but it has other purposes too. The GROUP keyword lets you define an array, each of whose elements is the specified group. Thus a GROUP name followed by a bounds list specifies that there are multiple occurrences of a GROUP within a RECORD. For example:

```
1000    RECORD FAMILY

                GROUP PARENTS
                        STRING FATHER = 30
                        STRING MOTHER = 30
                END GROUP PARENTS

                WORD KIDS_COUNT

                GROUP KIDS(10)
                        STRING KID = 30
                        DECIMAL(2,0) AGE
                END GROUP KIDS

        END RECORD FAMILY
```

This program specifies 11 instances of GROUP KIDS in RECORD FAMILY.

Once the record template has been defined, you can declare an instance of it:

```
2000    MAP (FAM) FAMILY MY_FAMILY
```

In this case, the GROUP name is required when referencing any of the group components because the GROUP name is an array. Thus, MY_FAMILY::AGE is an ambiguous reference because no element of GROUP KIDS is specified. An unambiguous reference would be MY_FAMILY::KIDS(3)::AGE.

Line 2000 allocates 414 bytes of storage: 30 bytes each for strings FATHER and MOTHER, two bytes for the WORD integer KIDS_COUNT, and 352 bytes for GROUP KIDS (30 bytes for string KID and 2 bytes for the DECIMAL number AGE, multiplied by 11). Because this storage is allocated with the MAP statement, you can use it as a file's record buffer by naming this map in an OPEN statement's MAP clause.

6.1.2 RECORD Variants

In some cases it is useful to have different record components overlay the same record field. This usage is called a record variant. A record component outside of the overlaid fields usually determines which record variant is being used in a particular reference. For example:

```
1000    RECORD EMP_WAGE_CLASS
            STRING NA.ME = 30
            STRING STREET = 15
            STRING CITY = 20
            STRING STATE = 2
            DECIMAL(5,0) ZIP
            STRING WAGE_CLASS = 1

            VARIANT
                    CASE
                      GROUP HOURLY
                        DECIMAL(4,2) HOURLY_WAGE
                        SINGLE REGULAR_PAY_YTD
                        SINGLE OVERTIME_PAY_YTD
                      END GROUP HOURLY
                    CASE
                      GROUP SALARIED
                        DECIMAL(7,2) YEARLY_SALARY
                        SINGLE PAY_YTD
                      END GROUP SALARIED
                    CASE
                      GROUP EXECUTIVE
                        DECIMAL(8,2) YEARLY_SALARY
                        SINGLE PAY_YTD
                        SINGLE EXPENSES_YTD
                      END GROUP EXECUTIVE
            END VARIANT
        END RECORD EMP_WAGE_CLASS

1500    DECLARE EMP_WAGE_CLASS EMP

2000    LINPUT "Name"; EMP::NA.ME
        LINPUT "Street"; EMP::STREET
        LINPUT "City"; EMP::CITY
        LINPUT "State"; EMP::STATE
        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
```

The value of the WAGE_CLASS component determines the format of the variant part of the record. Thus, WAGE_CLASS is called a "variant tag."

When BASIC allocates space for a variant RECORD, the amount allocated is for the case requiring the most storage. Variant fields can appear anywhere within the RECORD.

The previous example uses the CASE statement to assign data to record variants. The program prompts for those record components that are the same in all variants. 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.

6.1.3 Accessing RECORD Components

You will often want to access many or all components of a record. Because it is cumbersome to specify all the intermediate components each time you reference a record component, BASIC allows *elliptical references* to RECORD components.

An elliptical reference is one that does not explicitly include all the intermediate RECORD components. For example:

```
1000    RECORD YACHT

                GROUP TYPE_OF_YACHT
                        STRING MANUFACTURER = 10
                        STRING MODEL = 10
                END GROUP TYPE_OF_YACHT

                GROUP SPECIFICATIONS
                        STRING RIG = 6, LENGTH_OVER_ALL = 3
                        DECIMAL(5,0) DISPLACEMENT
                        DECIMAL(2,0) BEAM
                        DECIMAL(7,2) PRICE
                END GROUP SPECIFICATIONS

        END RECORD YACHT

2000    DECLARE YACHT FLEET(100)
        DECLARE INTEGER I
2100    FOR I = 0 TO 100
                        LINPUT 'Model'; FLEET(I)::MODEL
                        LINPUT 'Manufacturer'; FLEET(I)::MANUFACTURER
                        INPUT 'Rig'; FLEET(I)::RIG
                        INPUT 'Overall length'; FLEET(I)::LENGTH_OVER_ALL
                        INPUT 'Displacement'; FLEET(I)::DISPLACEMENT
                        INPUT 'Beam'; FLEET(I)::BEAM
                        INPUT 'Price'; FLEET(I)::PRICE
2300    NEXT I
```

In the FOR–NEXT loop, the references to record components do not include any GROUP names because the elementary component names are unambiguous.

The rules for using elliptical references are as follows:

1. The RECORD instance must always be specified.
2. Any dimensioned GROUP name must always be specified.
3. Any other intermediate component name may be omitted.
4. The final component name must be specified.

For example:

```
10        RECORD A
                  INTEGER B(2)
                  GROUP C1
                    REAL D
                    STRING E
                  END GROUP
                  GROUP C2(5)
                    INTEGER D
                    DECIMAL F(4)
                  END GROUP
          END RECORD

20        DECLARE A X(10), Y
```

Line 10 defines the RECORD and line 20 declares instances of this RECORD template. The minimal reference to the string E in record instance X is:

X(i)::E

In this example, i is the index of array X. The minimal reference to string E in record instance Y is:

Y::E

The minimal reference for the REAL variable D in record instance X is:

X(i)::C1::D

In this example, i is the index of array X. The minimal reference to the REAL variable D in record instance Y is:

Y::C1::D

Because there is a D in groups C1 and C2, you must specify either C1::D or C2(i)::D. In this case, extra component names are required to resolve an otherwise ambiguous reference. The minimal reference to DECIMAL F in record instances X and Y is the fully qualified references:

X(i)::C2(j)::F

Y::C2(j)::F

In this example, i is an index into array X and j is an index into the GROUP array C2 in record instance Y.

Even though F is a unique field name within record A, the element of array C2 must be specified. In this case the extra components must be specified due to subscripting.

You can assign all the values from one record instance to another, as long as the record instances are identical except for names. For example:

```
10        RECORD x
              GROUP y(4)
                  STRING z=10
                  REAL zz
                  INTEGER zzz
              END GROUP
          END RECORD
```

(continued on next page)

```
20        RECORD a
              GROUP b
                  STRING c=10
                  REAL cc
                  INTEGER ccc
              END GROUP
          END RECORD
30        RECORD i
              STRING j=10
              REAL jj
              INTEGER jjj
          END RECORD
40        DECLARE X x1,x2,x3(3)
          DECLARE A a1
          DECLARE I i1,i2(10)
50        x1 = x2
          x2 = x3(2)
          x3(3) = x1
```

Record instances X1, X2, and the individual elements of array X3 have the same form, that is, an array of 4 GROUPS, each of which contains a 10–byte string followed by a REAL followed by an INTEGER. Any of these record instances can be assigned to one another.

However, X3 and X1 do not have the same form because X3 is an array of RECORD x and X1 is a single instance of RECORD x. Thus, X1 and X3 cannot be assigned to one another.

Chapter 7
ANSI Minimal BASIC

This chapter explains the operation of the VAX–11 BASIC compiler when used with the /ANSI_STANDARD qualifier.

7.1 Introduction

The American National Standard for Minimal BASIC (ANSI X3.60–1978) describes a nucleus of the BASIC programming language. This nucleus will be a part of any BASIC implementation that conforms to this standard. Thus, writing programs that conform to the ANSI Minimal BASIC standard helps assure that they will run under any standard implementation of BASIC.

The ANSI Minimal BASIC Standard allows both extensions to the current standard and features whose behavior is defined by each implementation. This chapter describes these extensions and implementation-defined features. Many features of VAX–11 BASIC are allowed as extensions to ANSI Minimal BASIC. For example, programs with 31–character variable names will compile correctly; however, BASIC reports an informational message for each instance of a long variable name. This tells you that your program does not strictly conform to ANSI Minimal BASIC.

Note that certain features of VAX–11 BASIC are invalid in ANSI Minimal BASIC programs. For example, variables ending in a percent sign are invalid because ANSI Minimal BASIC does not allow integer variables. If a VAX–11 BASIC feature is invalid in ANSI Minimal BASIC, BASIC signals the fatal error "INTDATTYP, integer data type not supported in ANSI".

For a thorough understanding of ANSI Minimal BASIC, you should obtain and read ANSI X3.60–1978.

Note that the descriptions and explanations in this chapter apply only to programs compiled or run with the /ANSI_STANDARD qualifier in effect.

7.2 /ANSI_STANDARD Qualifier

/ANSI_STANDARD is both a qualifier to the DCL BASIC command and to the SET command in the BASIC environment. When you specify this qualifier, the following qualifiers are not allowed:

- /TYPE_DEFAULT

The default data type is always REAL. However, you can use a /REAL_SIZE qualifier to specify SINGLE, DOUBLE, GFLOAT, or HFLOAT floating-point numbers.

- /NOSETUP
- /NOLINE
- /FLAG

BASIC signals a fatal error if you use any of these qualifiers in addition to the /ANSI_STANDARD qualifier.

Also, you cannot use compiler directives in programs compiled with the /ANSI_STANDARD qualifier.

7.3 Extensions to X3.60–1978

The following items are extensions to the ANSI Minimal BASIC Standard. In order to write completely transportable programs, you should avoid these extensions and use only the capabilities required by the standard. Note that VAX–11 BASIC reports an informational error if you use any extensions.

7.3.1 Statements

The ANSI Minimal BASIC Standard requires that each program have an END statement. Also, the LET keyword is not optional in ANSI Minimal BASIC. BASIC signals "LETKEYREQ, LET keyword required in ANSI" when it encounters an assignment statement without the LET keyword. BASIC signals "ENDSTAREQ, END statement required in ANSI" if your program does not have an END statement.

7.3.2 Variables

The ANSI Minimal BASIC Standard requires only 2–character names for numeric variables and arrays and 1–character names followed by a dollar sign for string variables and arrays. With the /ANSI_STANDARD qualifier in effect, VAX–11 BASIC allows up to 31–character variable and array names. Names ending in a percent sign (%) are invalid as are any explicitly declared variables.

All VAX–11 BASIC keywords remain reserved words when the /ANSI_STANDARD qualifier is in effect; you cannot use these reserved words as variable names. See Appendix A in the *BASIC User's Guide* for a list of reserved keywords.

The ANSI Minimal BASIC Standard requires the LET keyword when assigning values to variables. In VAX–11 BASIC, the LET keyword is optional even with the /ANSI_STANDARD qualifier in effect.

7.3.3 Numeric Constants

The ANSI Minimal BASIC Standard allows numeric constants of the form:

sd...d	(implicit point representation)
sd...drd...d	(explicit point unscaled representation)
sd...drd...dEsd...d	(explicit point scaled representation)
sd...dEsd...d	(implicit point scaled representation)

where:

- d Is a decimal digit.
- r Is a period.
- s Is an optional sign.
- E Is the explicit character E.

In addition to constants of this form, VAX–11 BASIC with the /ANSI_STANDARD qualifier in effect allows: 1) integer constants that end in a percent sign (%) and 2) explicitly typed numeric constants. See Chapter 5 in the *BASIC User's Guide* for more information about explicitly typed numeric constants.

7.3.4 User-Defined Functions (DEF Statement)

The ANSI Minimal BASIC Standard requires that user-defined functions accept a single parameter. The formal parameter in the DEF must be an unsubscripted numeric variable. With the /ANSI_STANDARD qualifier in effect, VAX–11 BASIC reports a syntax error for DEFs that specify more than one parameter.

The ANSI Minimal BASIC Standard makes no mention of DEFs with string parameters. VAX–11 BASIC in /ANSI_STANDARD mode allows string DEFs but signals the informational error "STRDEFNOT, string DEF not ANSI".

7.3.5 Built-In Functions

The ANSI Minimal BASIC Standard allows only the following implementation-supplied (or built-in) functions:

- ABS
- ATN
- COS
- EXP
- INT
- LOG
- RND
- SGN
- SIN
- SQR
- TAN

With the /ANSI_STANDARD qualifier in effect, BASIC reports an informational error "FEANOTANS, language feature not ANSI" if you use any VAX–11 BASIC built-in functions other than these. Further, the following functions are not allowed:

- DECIMAL
- INTEGER
- REAL
- LOC
- GETRFA

BASIC reports a fatal error if you use any of these functions in a program compiled with the /ANSI_STANDARD qualifier.

7.3.6 Arrays

The ANSI Minimal BASIC Standard requires that array declarations are valid only for numerics and allow one or two dimensions. All arrays have a lower bound of zero unless an OPTION BASE statement specifies a lower bound of one. The format of OPTION BASE is:

```
OPTION BASE n
```

where:

n   Is either zero or one. Zero specifies that the lower bound of arrays is either (0) or (0,0). One specifies that the lower bound is either (1) or (1,1).

With the /ANSI_STANDARD qualifier in effect, VAX–11 BASIC allows arrays of any data type, including strings. However, an attempt to use an array having more than two dimensions causes BASIC to signal "FEANOTANS, language feature not ANSI".

Although you can have arrays of any floating-point data type, you control this feature with qualifiers to the DCL BASIC command or the SET command. This means that all floating-point arrays in a single program are of the same data type.

7.3.7 Program Format

With the /ANSI_STANDARD qualifier in effect, VAX–11 BASIC allows comment fields beginning with an exclamation point and ending with an exclamation point or a carriage return. VAX–11 BASIC also allows explicit line continuation with ampersands and backslashes. However, implicit line continuation is invalid.

7.4 Implementation-Defined Features

The American National Standard for Minimal BASIC leaves the following features to be defined by the implementation. In each case, the behavior of these implementation-defined features is as described for VAX–11 BASIC:

- Accuracy of evaluation of numeric expressions

  The standard does not specify a minimum accuracy, but recommends at least six significant decimal digits of precision. In VAX–11 BASIC, the accuracy of numeric expressions is always the same as that of the operands. This is specified with the /REAL_SIZE qualifier.

- End-of-input-reply

  In VAX–11 BASIC, the end-of-input-reply is a carriage return.

- End-of-print-line

  In VAX–11 BASIC, the end-of-print-line is a carriage return/line feed combination (ASCII code 13 and 10).

- Exrad-width for printing numeric representations

  The standard requires at least two digits for the representation of exponents. For programs compiled with /REAL_SIZE of SINGLE or DOUBLE, exrad-width is two. For programs compiled with /REAL_SIZE=GFLOAT, exrad-width is three. For programs compiled with /REAL_SIZE=HFLOAT, exrad-width is four.

- Initial values for variables

  The standard recommends that all variables are "detectably undefined in the sense that an exception will result from any attempt to access the value of a variable before that variable is explicitly assigned a value." Therefore you should explicitly initialize all variables. VAX–11 BASIC initializes all numeric variables to zero and all dynamic string variables to the null string.

- Input-prompt

  The standard recommends that the input-prompt be a question mark followed by a single space. VAX–11 BASIC conforms to this recommendation. Note that you cannot supply a string constant to be displayed as an input-prompt. If you attempt to supply a string prompt, BASIC signals "FEANOTANS, language feature not ANSI".

- Longest string that can be retained

  The standard states that string variables must be able to contain strings of at least 18 characters. VAX–11 BASIC lets you use strings of up to 65535 characters.

- Machine infinitesimal

  The standard recommends that machine infinitesimal be at most 1E–38. For programs compiled with /REAL_SIZE of SINGLE or DOUBLE, machine infinitesimal is approximately 2.9E–39, with /REAL_SIZE=GFLOAT machine infinitesimal is approximately 5.6E–308, with /REAL_SIZE=HFLOAT machine infinitesimal is approximately 8.4E–4933.

- Machine infinity

  The standard recommends that machine infinity be at least 1E38. For programs compiled with /REAL_SIZE of SINGLE or DOUBLE, machine infinity is approximately 1.7E38, with /REAL_SIZE=GFLOAT machine infinity is approximately 9.0E309, with /REAL_SIZE=HFLOAT machine infinity is approximately 8.4E4933.

- Margin for output line

  The standard makes no recommendation for the width of the output line. With the /ANSI_STANDARD qualifier in effect, the margin width for the controlling terminal is 80 characters. Note that the margin width for the controlling terminal is infinite for programs compiled /NOANSI_STANDARD.

- Precision for numeric values

  The standard recommends at least six significant decimal digits of precision. This corresponds to the SINGLE argument of the VAX–11 BASIC REAL_SIZE qualifier. You can also specify DOUBLE or GFLOAT (up to 15 significant decimal digits of accuracy), or HFLOAT (up to 33 significant decimal digits of accuracy). See Chapter 5 in the *BASIC User's Guide* for more information.

  Note that the accuracy of numeric expressions is always the same as the precision specified with an argument to the REAL_SIZE qualifier.

- Print zone length

  VAX–11 BASIC always has five 14–position print zones per print line.

- Pseudorandom number sequence

  The RND function produces a pseudorandom sequence of numbers until the RANDOMIZE statement is executed. After RANDOMIZE executes, the RND function produces a random sequence of numbers.

- Significance-width for printing numeric representations

  The standard specifies at least six positions. In VAX–11 BASIC, the PRINT statement provides up to six positions for numeric values, regardless of the floating-point data type in effect.

Chapter 8
I/O on VAX/VMS

This chapter describes some of the more advanced I/O features available in VAX–11 BASIC. For more information on I/O to RMS disk files, see the *BASIC User's Guide*. Subjects covered in this chapter are:

- VAX/VMS logical names
- RMS I/O to ANSI magnetic tapes
- Device-specific I/O to magnetic tapes, disks, and unit record devices
- I/O to mailboxes
- Network I/O
- File sharing and explicit record locking

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:

```
100     OPEN "MTA2:" FOR OUTPUT AS FILE #1
200     OPEN "MTA1:PARTS.DAT" FOR INPUT AS FILE #2, SEQUENTIAL
```

Because the file specification does not contain a file name, line 100 opens the tape drive for device-specific I/O. On the other hand, line 200 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 disks devices.

8.1 Using Logical Names in File Specifications

Logical names let you 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 need not depend on the availability of a specific device. You can define logical names:

1. From DCL command level with the ASSIGN command
2. From within a program with the SYS$CRELOG system service
3. From within the BASIC environment with the BASIC ASSIGN command

VAX–11 BASIC supports any valid logical name as part of a file specification.

A logical name specifies a 1– through 63–character name to be associated with the specified device or file specification. If the logical name specifies a device, end the logical name with a colon. The following example defines a logical name for a file specification:

```
$ ASSIGN DBA1:[SENDER]PAYROL.DAT PAYR
```

This example defines a logical name for a physical device:

```
$ ASSIGN DBA2: DISK:
```

Once you define the logical name, you can reference that name in your program. For example:

```
10      OPEN "PAYR" FOR INPUT AS FILE #1%, &
              ORGANIZATION SEQUENTIAL
20      OPEN "DISK" FOR OUTPUT AS FILE #2%, &
              ORGANIZATION VIRTUAL
```

These OPEN statements do not depend on the availability of DBA1: or DBA2: in order to work. If these devices are not available, you can simply redefine the logical names so that they specify other disk drives before running the program.

For complete information on logical names, see the *VAX/VMS Command Language User's Guide*.

8.2 RMS I/O to Magnetic Tape

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 (per magnetic tape) at a time, and the files are not available to other users.

8.2.1 Allocating and Mounting the 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:

```
$ MOUNT /DENSITY = 800 MT1: VOL001
```

If the records do not specify the size of the block (no value in HDR 2), specify BLOCKSIZE as part of the MOUNT command. For example:

```
$ MOUNT /DENSITY = 800/BLOCKSIZE = 128 MT1: VOL020
```

8.2.2 Opening FOR OUTPUT

You create and open the magnetic tape file for output with the syntax:

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

,[ORGANIZATION] SEQUENTIAL **[{ FIXED | VARIABLE }]**

[,MAP map-nam]
[,NOREWIND]
[,RECORDSIZE int-exp]
[,BLOCKSIZE int-exp]

For example:

```
40      OPEN "MT1:PARTS.DAT" FOR OUTPUT AS FILE #2%, SEQUENTIAL FIXED
                ,RECORDSIZE 256%, BLOCKSIZE 4%
```

This example opens the file "PARTS.DAT" and writes 256 byte records that are blocked four to a physical tape block of 1024 bytes.

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. 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, BASIC defaults to one record per block. For small records, this can be inefficient; the tape will contain many inter-record gaps.

8.2.3 Opening FOR INPUT

You open an existing magnetic tape file with the syntax:

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

,[ORGANIZATION] SEQUENTIAL **[{ FIXED | VARIABLE }]**

,ACCESS READ
[,MAP map-nam]
[,NOREWIND]
[,BLOCKSIZE int-exp]
[,RECORDSIZE int-exp]

For example:

```
100      OPEN "MT2:PAYROLL.DAT" FOR INPUT AS FILE #4%
                ,RECORDSIZE 1024%, BLOCKSIZE 2%, ACCESS READ
```

This statement opens the file "PAYROLL.DAT" and specifies 1024 byte records that are blocked two to each physical tape block. If you do not specify a record size or a block size, BASIC defaults to the values in the header block. If you do not specify a record format in the OPEN FOR INPUT clause, BASIC defaults to the format present in the header block. You must specify ACCESS READ to read any records written to the file.

8.2.4 Positioning the Tape (NOREWIND)

NOREWIND positions the tape for reading and writing as follows:

- Specifying NOREWIND when you create the 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, BASIC rewinds and continues the search until reaching the logical end-of-tape again. Omitting NOREWIND tells 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:

```
10       OPEN "MT1:PAYROL.DAT" FOR OUTPUT AS FILE #1% &
                ,ORGANIZATION SEQUENTIAL, NOREWIND
```

This 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 (BOT), logically deleting all subsequent files.

Note that you cannot specify REWIND; to avoid advancing the tape, simply omit the NOREWIND keyword.

8.2.5 Accessing ANSI Magnetic Tape Files

Because ANSI tape files are RMS sequential files, you write records with PUT statements and read records with GET statements.

8.2.5.1 Writing Records (PUT)

The PUT statement writes sequential records to the file. For example:

```
10       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
```

This program writes a record to the file. Successive PUTs write successive records.

Each PUT writes one buffer, or tape block, to the file. If your OPEN statement specifies a RECORDSIZE clause, the record buffer length equals RECORDSIZE. For example:

```
RECORDSIZE 60%
```

This clause specifies a record length and a record buffer size of 60 bytes. You can specify a record length between 18 and 8192 bytes. The default is 132 bytes.

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.

8.2.5.2 Reading Records (GET)

The GET statement reads one logical record into the buffer. For example:

```
240     OPEN "MTO: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
```

In this program, each GET reads a group of four records (a total of 80 bytes) from the file on channel 5. Successive GETs read successive 80–byte groups of records.

8.2.6 Speeding Access Time (BLOCKSIZE)

Magnetic tape block sizes range from 18 to 8192 bytes. With RMS tapes, you specify this size in the BLOCKSIZE clause as a positive integer indicating the number of records in each block. BASIC then calculates the actual size in bytes. Thus, a file on tape with 126 byte records can have a block size between 1 and 64. The default is 126 bytes (one record per block). For example:

```
10      OPEN "MTO:[SMITH]TEST.SEQ" FOR OUTPUT AS FILE #12% &
                ,ORGANIZATION SEQUENTIAL FIXED, RECORDSIZE 90% &
                ,BLOCKSIZE 12%
```

In this 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 moves 1080 bytes of data between the tape and this buffer. Every twelfth GET or PUT causes a physical read or write. The next eleven GETs or PUTs only move data into or out of the I/O buffer. Specifying a BLOCKSIZE larger than the default can reduce overhead by eliminating some physical reading and writing to the tape. In the example, a BLOCKSIZE of 12 saves time by accessing the tape only after every twelfth record operation.

8.2.7 Blocking Records

Through RMS, 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 GETs until it locates the fields of the record you want. For example:

```
110     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
```

This 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.

8.2.8 Rewinding the Tape (RESTORE)

With the RESTORE statement, you can rewind the tape to the start of the currently open file. For example:

```
10      OPEN "MTO:FTF.DAT" FOR INPUT AS FILE #2%, ACCESS READ
20      GET #2%
              !.
              !.
              !.
500     RESTORE #2%
510     GET #2%
```

You cannot rewind past the beginning of the currently open file.

8.2.9 Closing the File (CLOSE)

The CLOSE statement ends I/O to the file. For example:

```
590     CLOSE #6%
```

This statement ends input and output to the file open on channel 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), BASIC:

- Writes file trailer labels and two end-of-file marks following the last record
- Backspaces over the last end-of-file mark
- Releases allocated buffer space

BASIC does not rewind the tape.

8.3 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.

8.3.1 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:

```
50      MAP (DNG) A% = 80%
100     OPEN "CR1:" FOR INPUT AS FILE #1%, ACCESS READ, MAP DNG
110     GET #1%
```

BASIC treats the device as a file, and data is read from the card reader as a series of fixed-length records.

8.3.2 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 performing device-specific I/O.

8.3.2.1 Allocating and Mounting the Tape

You must allocate the tape unit to your process before starting file operations. For example:

```
$ ALLOCATE MT1:
```

This command assigns tape drive MT1: to your process.

Use the DCL MOUNT command and the "FOREIGN" qualifier (/FOR) to mount the tape. For example:

```
$ MOU /FOR MT1:
```

If your program needs: 1) a block size other than 512 bytes, or 2) a particular tape density, specify these characteristics with the MOUNT command as well. For example:

```
$ MOU /FOR /BLOCKSIZE = 1024/ DENSITY = 800 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.

8.3.2.2 Opening FOR OUTPUT

You create and open the magnetic tape file for output with the syntax:

```
OPEN "dev:" FOR OUTPUT AS FILE [#]chnl-exp

,[ORGANIZATION] SEQUENTIAL    [ VARIABLE ]
                              [ FIXED    ]

[,NOREWIND]
[,MAP map-nam]
[,RECORDSIZE int-exp]
[,USEROPEN program-nam]
```

For example:

```
190     OPEN "MT1:" FOR OUTPUT AS FILE #1%,   &
 ORGANIZATION SEQUENTIAL VARIABLE
```

This 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.

8.3.2.3 Opening FOR INPUT

You can open an existing magnetic tape file with the syntax:

```
OPEN "dev:" FOR INPUT AS FILE [#]chnl-exp

,ACCESS READ
[,NOREWIND]
[,MAP map-nam]
[,RECORDSIZE int-exp]
[,USEROPEN program-nam]
```

For example:

```
140     OPEN "MT2:" AS FILE #2%
```

This statement opens the file on tape unit MT2:.

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 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 will read one physical record whose size is returned in RECOUNT. If you are using a MAP only, the first n bytes (where "n" is the value returned in RECOUNT) are valid.

8.3.2.4 Writing Records (PUT)

The PUT statement writes records to the file in sequential order. For example:

```
10      OPEN "MT0:" FOR OUTPUT AS FILE #9%, &
             SEQUENTIAL VARIABLE
30      INPUT "NAME";NA.ME$
40      MOVE TO #9%, NA.ME$
50      PUT #9%
```

Line 50 writes the contents of the buffer to the file. Successive PUTs write successive records.

The default record length (and therefore, the size of the buffer) is 132 bytes. The RECORDSIZE attribute tells BASIC to read or write records of a specified length. For example:

```
100     OPEN "MT0:" FOR INPUT AS FILE #1%, RECORDSIZE 900%
```

This statement opens tape unit MT0: and specifies records of 900 characters. You must specify an even integer larger than 18. If you specify a buffer length less than 18, BASIC signals an error. If you try to PUT a record longer than the buffer, BASIC signals the error "Size of record invalid" (ERR = 156).

To write records shorter than the buffer, include the COUNT clause with the PUT statement. For example:

```
50      PUT #6%, COUNT 56%
```

This statement writes a 56 character record to the file open on channel 6. If you do not specify COUNT, 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 BASIC nor RMS prefixes the records with any count bytes.

8.3.2.5 Reading Records (GET)

The GET statement reads records into the buffer. For example:

```
50      OPEN "MT1:" FOR INPUT AS FILE #1%, ACCESS READ
60      GET #1%
70      MOVE FROM #1%, A$ = RECOUNT
80      PRINT A$
90      RESTORE #1%
100     CLOSE #1%
```

This program reads a record into the buffer, prints a string field, and rewinds the file before closing. Successive GETs read successive records. BASIC signals the error "End of file on device" (ERR = 11)

if you encounter a tape mark during a GET. If you trap this error and continue, you can skip over any tape mark(s). The system variable RECOUNT is set to the number of bytes transferred after each GET.

8.3.2.6 Rewinding the Tape (RESTORE)

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:

```
190     OPEN "MT1:" FOR OUTPUT AS FILE #2%, ACCESS READ
                !.
                !.
                !.
550     PUT #2%
560     RESTORE #2%
570     INPUT "NEXT RECORD"; NXTRECBB%
```

If you rewind a tape opened without ACCESS READ before closing it, you erase all data written before the RESTORE.

8.3.2.7 Closing the Tape (CLOSE)

The CLOSE statement ends I/O to the tape. For example:

```
300     CLOSE #12%
```

This statement ends input and output to the tape open on channel 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), BASIC:

- Writes file trailer labels and two end-of-file marks following the last record
- Backspaces over the last end-of-file mark
- Releases allocated buffer space

The tape is not rewound unless you specified RESTORE in your program.

8.3.3 Device-Specific I/O to Disks

The following sections describe device-specific I/O to disks.

8.3.3.1 Assigning and Mounting the Disk

You must allocate a disk unit to your process before starting operations. For example:

```
$ ALLOCATE DBA3:
```

This command assigns disk DBA3: to your process.

When performing I/O directly to a disk, you must mount the disk with the MOUNT command before accessing it. For example:

```
$ MOU /FOR DBA3:
```

You can then open the disk for input or output.

8.3.3.2 Opening for OUTPUT

You create and open the disk file with the syntax:

OPEN "dev:" FOR OUTPUT AS FILE [#]chnl-exp

,[ORGANIZATION] VIRTUAL
[,MAP map-nam]
[,RECORDSIZE int-exp]
[,USEROPEN program-nam]

For example:

```
1000    OPEN "DBA3:" FOR OUTPUT AS FILE #2%, ORGANIZATION VIRTUAL
```

You can then write data to the disk.

The recordsize determined by the MAP or RECORDSIZE clause must be an integer multiple of 512 bytes. The default is 512.

8.3.3.3 Opening for INPUT

You can open an existing disk file with the syntax:

OPEN "dev:" FOR INPUT AS FILE [#]chnl-exp

,[ORGANIZATION] VIRTUAL
,ACCESS READ
[,MAP map-nam]
[,RECORDSIZE int-exp]
[,USEROPEN program-nam]

For example:

```
1000    OPEN "DBA1:" FOR INPUT AS FILE #4%, ORGANIZATION VIRTUAL
```

You can then read data from the disk.

The recordsize 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: 1) you plan to read from the disk and 2) it is mounted read only.

8.3.3.4 Accessing the Disk

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.

8.3.3.5 Writing Records (PUT)

You write data by defining a record buffer and writing the data to the file with PUT statements. For example:

```
0900     INPUT "HOW MANY RECORDS TO WRITE"; J%
1000     OPEN "DBB2:" FOR OUTPUT AS FILE #2%, ORGANIZATION VIRTUAL
1005     FOR K% = 1% TO J%
1010       FOR I% = 0% TO 7%
1020           INPUT "NAME OF BOOK"; BOOK_NAME$
1030           INPUT "RETRIEVAL NUMBER"; RET_NUM%
1040           INPUT "SUBJECT AREA"; SUBJ$
1050           MOVE TO #2%, FILL$ = I% * 64%, BOOK_NAME$, RET_NUM%, SUBJ$
1060       NEXT I%
1070     PUT #2%
1080     NEXT K%
1090     CLOSE #2
```

This program writes four 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.

When you write records, neither BASIC nor RMS prefixes the records with any count bytes.

8.3.3.6 Reading Records (GET)

You read data by defining a record buffer and reading the data from the file with GET statements. After the data has been retrieved with a GET statement you can deblock the data with MOVE or REMAP statements. For example:

```
5        ON ERROR GOTO 19000
800      MAP (SAM) STRING PRT_ID = 32%, SINGLE MAFLD,     &
  LONG ADIR_OLDN, STRING REST = 472%
900      MAP DYNAMIC (SAM) PRT_ID, MAFLD, ADIR_OLDN
1000     OPEN "DBA1:" FOR INPUT AS FILE #2%, ORGANIZATION VIRTUAL, &
             ACCESS READ, MAP SAM
1010     WHILE 1% = 1%
1020         GET #2%
1025         FOR I% = 0% TO 11%
                 REMAP (SAM) STRING FILL(I% * 40%), PRT_ID, MAFLD, ADIR_OLDN
                 PRINT PRT_ID, MAFLD, ADIR_OLDN
             NEXT I%
1030     NEXT
19000    ON ERROR GOTO 0 IF ERR <> 11
         RESUME 32000
32000    CLOSE #2%
32767    END
```

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, 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 the *BASIC User's Guide* for more information on MAP, MAP DYNAMIC, and REMAP statements.

8.4 Input and Output 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 the TMPMBX or PRMMBX privilege to use mailboxes. Mailboxes are created and deleted by system services. For more information on using system services in BASIC programs, see Chapter 6.

Use the EXTERNAL statement to define the SYS$CREMBX system service that creates the mailbox. In BASIC programs, you create mailboxes as integer functions passing: 1) a channel argument and 2) a string literal or a logical name for the mailbox. For example:

```
100     EXTERNAL INTEGER FUNCTION SYS$CREMBX
110     SYS$STATUS% = SYS$CREMBX( ,CHAN%,,,,,"MARG")
```

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.

To access mailboxes, you create one process for the sending program and a second for the receiving program. Run the sending program first. For example:

Program One: Sending Program

```
1000    EXTERNAL INTEGER FUNCTION SYS$CREMBX
        SYS$STATUS% = SYS$CREMBX( ,CHAN%,,,,,"MARG")
        OPEN "MARG" FOR INPUT AS FILE #1%
        INPUT "WHAT IS THE PASSENGER NAME"; PASS.NAME$
        PRINT #1%, PASS.NAME$
        LINPUT #1%, CONFIRM.MSG$
        PRINT CONFIRM.MSG$
                !.
                !.
                !.
```

Program Two: Receiving Program

```
5       ON ERROR GOTO 19000
        MAP (RES) PASS.NAME$ = 32%
        OPEN "MARG" FOR INPUT AS FILE #1%
        LINPUT #1%, PASS.NAME$
        OPEN "RESER.LST" FOR INPUT AS FILE #2%, &
            ORGANIZATION INDEXED, MAP RES, ACCESS READ
        FIND #2%, KEY 0% EQ PASS.NAME$
        RECEIVING.MSG$ = "PASSENGER RESERVATION CONFIRMED"
        PRINT #1%, RECEIVING.MSG$
        GOTO 32000
19000   IF (ERR = 155) AND (ERL = 1030) &
        THEN
            RECEIVING.MSG$ = "RESERVATION DOES NOT EXIST"
            RESUME 1060
        ELSE
            ON ERROR GOTO 0
        END IF
32000   CLOSE #2%, #1%
32767   END
```

The first program requests a passenger name and sends it to the mailbox. The second program looks up the name in an indexed file. If the passenger name exists, the second program writes the confirmation message to the mailbox. If the passenger name does not exist, the error handler writes an alternate message. The first program then reads the mailbox and returns the result.

BASIC treats the mailbox as a sequential file. You write to the file with PRINT # or PUT, and read with INPUT #, LINPUT #, and GET. When the writing program closes the mailbox, the reading program receives the end-of-file error message.

Note

All mailbox operations are synchronous. Control does not pass back from the mailbox operation to your program until the other program completes its operations.

8.5 Network I/O

If your system supports DECnet–VAX facilities, and your computer is one of the nodes in a DECnet–VAX network, you can communicate with other nodes in the network with BASIC program statements. BASIC lets you:

- 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)

8.5.1 Remote File Access

To write or read files at a remote site, include the node name as part of the file specification. For example:

```
1000     OPEN "WESTON::DBA1:[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 double quotes and placing it between the node name and the double colon. For example:

```
10       OPEN 'WESTON"HOLT PASWRD"::DBA0:[HOLT.TMP]INDEXU.DAT;4' &
                        FOR INPUT AS FILE #1%, INDEXED, PRIMARY TEXT$
```

This file specification accesses the account [HOLT.TMP] on node WESTON by giving the username HOLT and the password PASWRD. After accessing the file, your BASIC program can read and write records as if the file were in your account.

8.5.2 Task-to-Task Communication

BASIC supports task-to-task communication if your account has networking privileges (NETMBX).

In this procedure:

1. You 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. You 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 line:

```
1000    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:

```
1000    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.

For example:

Local Program

```
900        MAP (SJK) MSG$ = 32%
1000       OPEN 'WESTON"DAVIS PSWRD"::"TASK = MARG"' &
                  FOR OUTPUT AS FILE #1%, SEQUENTIAL, MAP SJK
1010       LINPUT "WHAT IS THE CUSTOMER NAME"; MSG$
1020       PUT #1%
1030       GET #1%
1040       PRINT MSG$
1050       CLOSE #1%
1060       END
```

Remote Node Program

```
5          ON ERROR GOTO 19000
                  .
                  .
                  .
970        MAP (SJK) MSG$ = 32%
1005       OPEN "SYS$NET" FOR INPUT AS FILE #1%, SEQUENTIAL, &
                  MAP SJK
1015       GET #1%
1025       MAP (FIL) NAME$ = 32%, RESERVATION$ = 64%
1035       OPEN "RESER.DAT" FOR INPUT AS FILE #2%, &
                  INDEXED, PRIMARY NAME$, MAP FIL
1045       FIND #2%, KEY 0% EQ MSG$
1055       MSG$ = "NAME CONFIRMED"
1065       PUT #1%
                  .
                  .
                  .
19000      IF (ERR = 153%) AND (ERL = 1045%)
             THEN MSG$ = "ERROR IN NAME"
                 RESUME 1065
                  .
                  .
                  .
32000      CLOSE #2%, 1%
32767      END
```

The task-to-task communication ends when the files are closed.

See the *DECnet VAX User's Guide* and the *DECnet VAX System Manager's Guide* for more information.

8.6 File Sharing and Explicit Record Locking

File sharing and explicit record locking let you control file and record operations when more than one access stream is simultaneously accessing a file. Two things determine whether a file can be shared by another user: 1) the file's protection level, and 2) the ALLOW clause in the OPEN statement that first accesses the file.

When a program attempts to open a file, the VAX/VMS operating system first checks the intended operations on the file (ACCESS READ, WRITE, or MODIFY) against the file's protection level. If the intended operation would violate the protection level, VAX/VMS denies access to the file. For example, if the file is protected against write access and the program attempts to open the file with ACCESS MODIFY, VAX/VMS does not allow access to the file.

Once the VAX/VMS operating system allows a program access to a file, RMS checks whether the file is already open. If it is, the ALLOW clause specified by the first program accessing the file takes effect. This ALLOW clause determines how other programs can access the file.

Whenever a program opens a file, it must declare:

- The record operations it intends to perform on the file (FIND, GET, PUT, UPDATE, DELETE, and SCRATCH). This corresponds to the ACCESS clause.
- The operations it will allow other programs to perform on the file. These are:
  - Read-type — other programs may access the file for GET and FIND operations only.
  - Write-type — other programs may access the file for PUT, DELETE, UPDATE, SCRATCH, GET, and FIND operations.

  This corresponds to the ALLOW clause.

The ALLOW clause determines the operations that subsequent access streams are allowed to perform on a file that is already open:

- ALLOW NONE locks the file for exclusive use by this access stream. Any attempt to share the file results in a "FILIS_LOC, File is locked" error (ERR=138).
- ALLOW READ locks the file against write-type operations. Any attempt to open the file with ACCESS WRITE or ACCESS MODIFY results in a "FILIS_LOC, File is locked" error (ERR=138).
- ALLOW WRITE locks the file against DELETE and SCRATCH operations. Any attempt to open the file with ACCESS MODIFY results in a "FILIS_LOC, File is locked" error (ERR=138).
- ALLOW MODIFY allows unlimited access to the file.

Note that a subsequent access stream can be, but does not have to be, in the same program that performed the original file OPEN.

8.6.1 Explicit Record Locking

When you use RMS disk files, RMS automatically locks each record you access. This automatic lock remains in effect only until the program executes another I/O operation on the same channel. In addition to automatic record locking, you can specify that each record accessed remains locked until you explicitly unlock it with the UNLOCK or FREE statement or until you close the file. You do this with the UNLOCK EXPLICIT clause in the OPEN statement.

The VAX/VMS operating system imposes limits on the number of record locks a process (and its subprocesses) can have in effect at any time. If you expect to keep a large number of records locked, you should have the process ENQLM (enqueue limit) set to a large value; otherwise, your program may cause an "Exceeded enqueue limit" error. See the *VAX/VMS System Management and Operations Guide* for more information.

You enable explicit record locking by specifying an UNLOCK EXPLICIT clause in the OPEN statement. Its format is:

UNLOCK EXPLICIT

Once you have opened a file with UNLOCK EXPLICIT, a FIND or GET statement prevents other access streams from retrieving that record. However, you can control the type of lock by specifying an ALLOW clause on the FIND or GET statement. The format of an ALLOW clause on a GET or FIND statement is:

```
{ GET  }  ,ALLOW  { NONE   }
{ FIND }          { READ   }
                  { MODIFY }
```

where:

NONE Specifies no access to the record.

READ Specifies read access to the record. This means that other access streams can retrieve the record, but cannot PUT or UPDATE the record.

MODIFY Specifies both read and write access to the record. This means that other access streams can GET, PUT, DELETE, or UPDATE the record.

If you do not specify an ALLOW clause, the default is ALLOW NONE.

The ALLOW clause must follow any RECORD or KEY clause. That is, it must be the last clause specified on the GET or FIND. For example:

```
50        MAP (XYZ)    &
          STRING       &
              FIRST_NAME = 10, &
              LAST_NAME = 20, &
              ADDRESS = 30
100       OPEN "IND.DAT" AS FILE #1, ORGANIZATION INDEXED, MAP XYZ, &
                    PRIMARY KEY LAST_NAME, UNLOCK EXPLICIT
200       GET #1, KEY #0 EQ "JONES", ALLOW READ
          !.
          !.
          !.
```

BASIC signals "ILLALLCLA, illegal ALLOW clause" if you specify an ALLOW clause on a GET or FIND and the file was not opened with the UNLOCK EXPLICIT clause.

Note that if you delete a record in a file that has been opened with the UNLOCK EXPLICIT clause, the record still remains locked until you explicitly unlock it.

When another record access stream attempts to access a locked record, BASIC signals "RECBUCLOC, Record/bucket locked" (ERR=154). However, if the program needs only read access to the record, it can override the lock with the REGARDLESS clause. Its format is:

GET #chnl-exp, position-clause, REGARDLESS

where:

chnl-exp	Is the number of an open channel.
position-clause	Is a KEY or RECORD clause.

You should be careful when using the REGARDLESS clause because a record accessed this way may be in the process of being changed by another program.

For example, assume that the IND.DAT file is shared by many programs and that any of the records in the file may be locked. This program reads the file sequentially and copies each record to a sequential file:

```
10          ON ERROR GOTO 19000
50          MAP (XYZ)    &
            STRING       &
                FIRST_NAME = 10, &
                LAST_NAME = 20, &
                ADDRESS = 30
100         OPEN "IND.DAT" AS FILE #1, ORGANIZATION INDEXED, MAP XYZ, &
                    PRIMARY KEY LAST_NAME, UNLOCK EXPLICIT
200         OPEN "BAKUP.DAT" FOR OUTPUT AS FILE #2, SEQUENTIAL, MAP XYZ
300         WHILE 1% = 1%
                GET #1, REGARDLESS
                PUT #2, COUNT RECOUNT
            NEXT
            !.
            !.
            !.
19000       IF (ERR = 11) AND (ERL = 300)
            THEN PRINT "FINISHED"
            RESUME 32766
            ELSE ON ERROR GOTO 0
            END IF
32766       CLOSE #1, 2
32767       END
```

Thus, this program produces a copy of the latest version of each record.

8.7 Sample Programs

The following programs use many of the features described in this chapter. Program One creates a single tape file in EBCDIC using IBM OS labels. Program Two reads a similar single tape file.

Program One

```
1     ON ERROR GOTO 19000              ! Error trap

800     MAP (BUFF) LAB_ALL$=80
     !
     ! VOL1 record
     !
     MAP (BUFF) LABEL$=4, VOL_SER$=6, FILL$=69, COL_80$=1
     !
     ! HDR1/EOF1 record
     !
     MAP (BUFF) FILL$=4, DS_ID$=17, DS_SER$=6, VOL_SEQ$=4      &
     , DS_SEQ$=4, GEN_NUM$=4, VER_NUM$=2     &
     , CRE_DAT$=6, EXP_DAT$=6, SCRTY$=1, BLK_CNT$=6     &
     , SYS_COD$=13
     !
```

```
    ! HDR2/EOF2 record
    !
    MAP (BUFF) FILL$=4, REC_FMT$=1, BLK_LEN$=5, REC_LEN$=5     &
    , TAP_DENS$=1, DS_POS$=1, FILL$=19, C_CTRL$=1     &
    , FILL$=1, BLK_ATR$=1, FILL$=11, SPEC$=2     &
    !
    ! Space for the time of day algorithm
    !
    MAP (FOO) BIN.TIM%, FILL%, TIM.BUF$=11
    MAP (FOO) FILL$=8, DAY.MON$=6, FILL$=3, YEAR$=2
    MAP (FOO) FILL$=9, D.Y$=2
    !
    ! Input file buffer
    !
    MAP (BUF1) BUF_1$=80

1000    LINPUT "Device name"; DEV_NAME$      ! Ask user for device
    OPEN DEV_NAME$ FOR OUTPUT AS FILE #1%     &
    , SEQUENTIAL FIXED        ! Label are fixed     &
    , ACCESS WRITE          ! Tape mark on close     &
    , MAP BUFF

1010    LINPUT "Input file"; IN_FIL$
    OPEN IN_FIL$ FOR INPUT AS FILE #3%     &
    , SEQUENTIAL FIXED        ! Only use fixed recs     &
    , ACCESS READ     &
    , MAP BUF1

1020    !
    ! Write VOL1 label
    !
    LAB_ALL$ = "VOL1"
    LINPUT "Vol serial id"; VOL_SER_TMP$
    VOL_SER$ = VOL_SER_TMP$
    CALL LIB$TRA_ASC_EBC(LAB_ALL$,LAB_ALL$) ! Xlate to EBCDIC
    PUT #1%

1030    !
    ! Write HDR1 Label
    !
    LAB_ALL$ = "HDR1"
    DS_ID$ = IN_FIL$              ! Use file-name
    DS_SER$ = VOL_SER_TMP$
    VOL_SEQ$, DS_SEQ$, GEN_NUM$ = "0001"
    VER_NUM$ = "00"
    GOSUB 10000                   ! Calc creation date
    BLK_CNT$ = "000000"
    EXP_DAT$ = " 00000"               ! No expiration date
    SCRTY$ = "0"                  ! No security
    SYS_COD$ = "DECFIL112"            ! DEC VAX/VMS Files-11
    HDR1_COPY$ = LAB_ALL$
    CALL LIB$TRA_ASC_EBC(LAB_ALL$,LAB_ALL$)
    PUT #1%

1040    !
    ! Write HDR2 label
    !
    LAB_ALL$ = "HDR2"
    INPUT "Blocking factor";BF%
    REC_LEN% = 80%                ! Use only 80 bytes
    BLK_LEN% = REC_LEN% * BF%
    TAP_DENS$ = "2"                 ! 800 BPI
    DS_POS$ = "0"
    RSET REC_LEN$ = STR$(1000000%+REC_LEN%) ! Leading zeroes
    RSET BLK_LEN$ = STR$(1000000%+BLK_LEN%) ! Ditto
    BLK_ATR$ = "B"                ! Blocked
```

(continued on next page)

```
    REC_FMT$ = "F"                  ! Fixed
    C_CTRL$ = " "                   ! No control
    HDR2_COPY$ = LAB_ALL$
    CALL LIB$TRA_ASC_EBC(LAB_ALL$,LAB_ALL$)
    PUT #1%

1060    CLOSE #1%                   ! Write the tape mark


2000    BLK_CNT% = 0%
    EOF_FLG% = 0%
    OPEN DEV_NAME$ FOR OUTPUT AS FILE #1%     &
    , SEQUENTIAL VARIABLE     &
    , ACCESS WRITE     ! Tape mark on close   &
    , RECORDSIZE BLK_LEN%     &
    , NOREWIND     ! After 1st tape mark

2020     BLK_LEN% = 0%

2030    FOR I% = 0% TO BF% - 1%
        GET #3%
        CALL LIB$TRA_ASC_EBC(BUF_1$,BUF_1$)
        MOVE TO #1%, FILL$=BLK_LEN%, BUF_1$
        BLK_LEN% = BLK_LEN% + REC_LEN%
    NEXT I%

2040     PUT #1%, COUNT BLK_LEN%
    BLK_CNT% = BLK_CNT% + 1%
    GOTO 2020 IF EOF_FLG% = 0%

3000    PRINT "End of file"
    OPEN DEV_NAME$ FOR OUTPUT AS FILE #1%     &
    , SEQUENTIAL FIXED     ! Trailer labels fixed     &
    , ACCESS WRITE         ! Tape mark on close     &
    , MAP BUFF     &
    , NOREWIND

3010    !
    ! Write EOF1 label
    !
    LAB_ALL$ = HDR1_COPY$
    LABEL$ = "EOF1"
    RSET BLK_CNT$ = STR$(1000000%+BLK_CNT%)     ! Actual count
    CALL LIB$TRA_ASC_EBC(LAB_ALL$,LAB_ALL$)
    PUT #1%

3020    !
    ! Write EOF2 label
    !
    LAB_ALL$ = HDR2_COPY$
    LABEL$ = "EOF2"
    CALL LIB$TRA_ASC_EBC(LAB_ALL$,LAB_ALL$)
    PUT #1%

4000  GOTO 32000

10000    !
    ! Routine to calc Julian date " YYDDD"
    !
    CALL SYS$ASCTIM(,TIM_BUF$,,)     ! Todays date
    DAY.MON$ = "01-JAN"     ! Want 1st of year
    CALL SYS$BINTIM(TIM.BUF$,BIN.TIM%)     ! 1-JAN this year
    CALL LIB$DAY(DAY.NUM%,BIN.TIM%)     ! Get day number 1-JAN
    CALL LIB$DAY(TODAY%)     ! Get day number today
```

```
DAY.NUM% = TODAY% - DAY.NUM% + 10001%     ! Dif with leading zero
DAY.MON$ = NUM$(DAY.NUM%)
D.Y$ = YEAR$
CRE_DAT$ = DAY.MON$
RETURN

19000   IF ERR = 11% THEN         ! If EOF on source ...
      IF ERL = 2025% THEN
          EOF_FLG% = -1%          ! Set flag
          RESUME 2040

19010   ON ERROR GOTO 0                ! Abort otherwise

32000  CLOSE #1%, 2%

32767  END
```

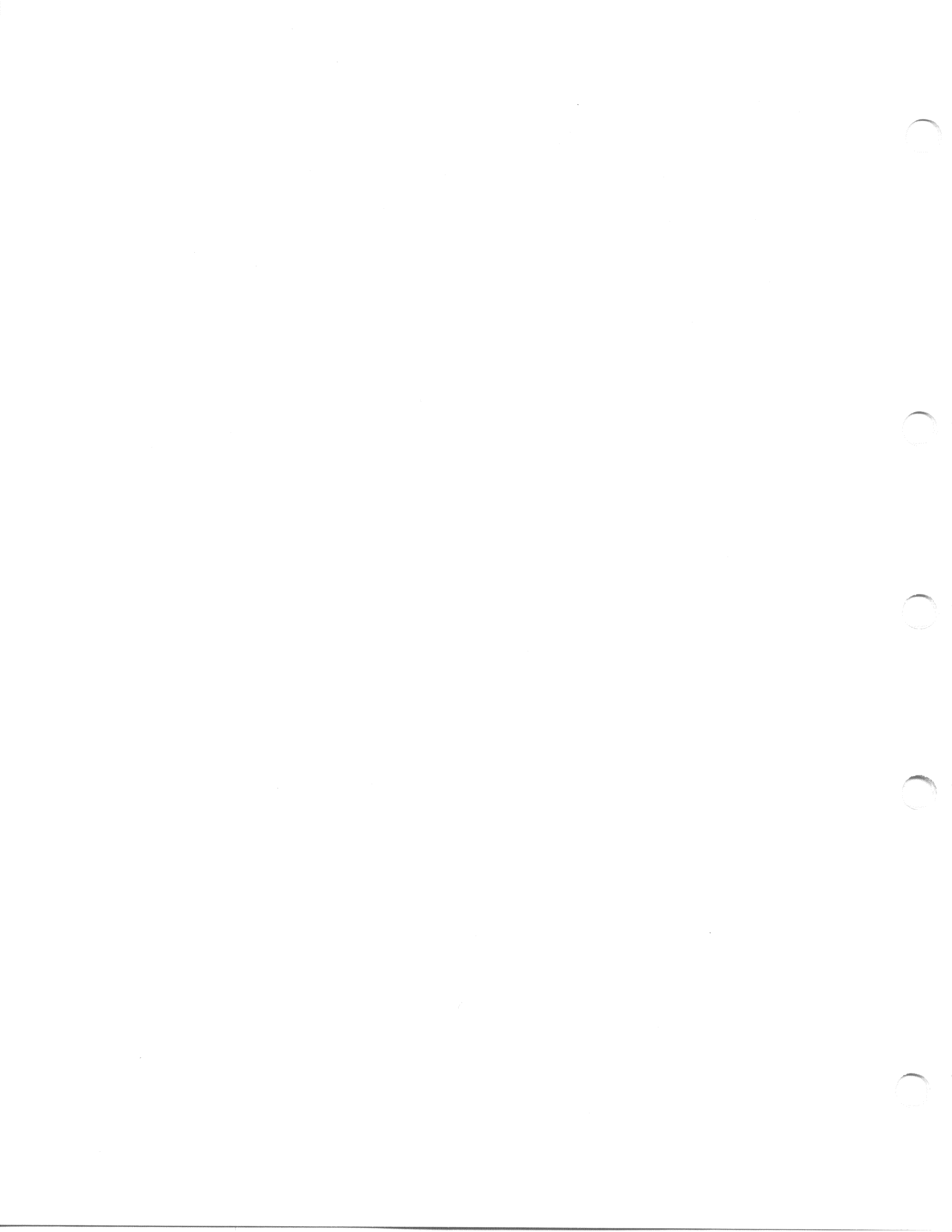
Program Two

```
1     ON ERROR GOTO 19000

400    DECLARE STRING DEV_NAME, OUT_FIL, EXPECT_LAB, TEMP
    DECLARE INTEGER REC_LEN_N, BLK_LEN_N

800    MAP (BUFF) STRING LAB_ALL=80
    !
    ! VOL1 record
    !
    MAP (BUFF) STRING LABEL=4, VOL_SER=6, FILL=69, COL_80=1
    !
    ! HDR1/EOF1 record
    !
    MAP (BUFF) STRING FILL=4, DS_ID=17, DS_SER=6, VOL_SEQ=4     &
    , DS_SEQ=4, GEN_NUM=4, VER_NUM=2     &
    , CRE_DAT=6, FILL=7, BLK_CNT=6, SYS_COD=13     &
    !
    ! HDR2/EOF2 record
    !
    MAP (BUFF) STRING FILL=4, REC_FMT=1, BLK_LEN=5, REC_LEN=5     &
    , FILL=21, C_CTRL=1, FILL=1, BLK_ATR=1     &
    , FILL=11, SPEC=2

1000    LINPUT "Device name"; DEV_NAME     ! Ask user for drive
    OPEN DEV_NAME FOR INPUT AS FILE #1%          &
    , SEQUENTIAL     ! Use default     &
    , ACCESS READ     ! No write to tape     &
    , MAP BUFF

1020    !
    ! Process the labels
    !
    EXPECT_LAB = "VOL1"
    GOSUB 10000
    PRINT "VOL serial number is "; VOL_SER

1030    EXPECT_LAB = "HDR1"
    GOSUB 10000
    PRINT "Data set ident is "; DS_ID
    PRINT "Data set serial is "; DS_SER
    PRINT "Volume sequence is "; VOL_SEQ
    PRINT "Creation date is  "; CRE_DAT
    PRINT "System code is   "; SYS_COD
```

(continued on next page)

```
1040    EXPECT_LAB = "HDR2"
    GOSUB 10000
    PRINT "Block length is  "; BLK_LEN
    PRINT "Record length is  "; REC_LEN
    PRINT "Record format is  "; REC_FMT
    PRINT "Block attribute is "; BLK_ATR
    BLK_LEN_N = VAL%(BLK_LEN)
    REC_LEN_N = VAL%(REC_LEN)

1060    GET #1%     ! Skip over tape-mark
    GOTO 1060    ! ...  any user labels

1080    CLOSE #1%

2000    !
    ! Now open the tape with the proper blocksize and read data
    !
    OPEN DEV_NAME FOR INPUT AS FILE #1%     &
    , SEQUENTIAL    &
    , ACCESS READ     ! No write to tape     &
    , RECORDSIZE BLK_LEN_N     ! Use supplied size     &
    , NOREWIND     ! Read in place

2010    LINPUT "Output file";OUT_FIL
    OPEN OUT_FIL FOR OUTPUT AS FILE #2%     &
    , SEQUENTIAL FIXED     &
    , ACCESS WRITE     &
    , RECORDSIZE REC_LEN_N     &
    , DEFAULTNAME ".DAT"

2020    GET #1%     ! Loop, reading blocks
    REC_CNT% = REC_CNT% + 1%

2030    ACT_REC_SIZ% = RECOUNT     ! Pick up actual size
    NUM_REC% = ACT_REC_SIZ% / REC_LEN_N     ! Partial block
    FOR I% = 0% TO NUM_REC% - 1%
        MOVE FROM #1%, FILL$=REC_LEN_N*I%, TEMP=REC_LEN_N
        CALL LIB$TRA_EBC_ASC(TEMP,TEMP)     ! Translate in place
        MOVE TO #2%, TEMP=REC_LEN_N     ! Deblock
        PUT #2%
    NEXT I%
    GOTO 2020

3000    !
    ! End of file.  Process EOF labels.
    !
    PRINT "End of file"
    OPEN DEV_NAME FOR INPUT AS FILE #1%     &
    , SEQUENTIAL     &
    , ACCESS READ     ! No write to tape     &
    , MAP BUFF     &
    , NOREWIND

3010    EXPECT_LAB = "EOF1"
    GOSUB 10000
    PRINT "*** BLOCK COUNT NOT MATCHED" IF REC_CNT% <> VAL%(BLK_CNT)

4000    GOTO 32000

10000    GET #1%     ! Process a label
    CALL LIB$TRA_EBC_ASC(LAB_ALL,LAB_ALL)
    RETURN IF LABEL = EXPECT_LAB
    PRINT "*** ERROR READING "; EXPECT_LAB; " RECORD"
    GOTO 32000
```

```
19000     !
    ! Error handler
    ! &
    IF ERR = 11% THEN     ! If EOF ...
        IF ERL = 1060% THEN     ! ...  on tape mark skip
            RESUME 1080     ! Then continue on
        ELSE IF ERL = 2020% THEN     ! ...  if on data read
        RESUME 3000     ! Then process EOF

19010     ON ERROR GOTO 0

32000     CLOSE #1%, 2%

32767     END
```



Chapter 9
The VAX–11 Common Data Dictionary

This chapter describes how to extract record definitions from the VAX–11 Common Data Dictionary.

9.1 Overview

The VAX–11 Common Data Dictionary (CDD) provides a way to specify record definitions that can be shared by many VAX/VMS layered products. The advantages to using the CDD are:

- Record declarations are language-independent
- A single declaration helps guarantee the accuracy and reliability of data
- The CDD maintains a history of users accessing it

When you extract a record definition from the CDD, BASIC translates the CDD definition to BASIC RECORD statement syntax, just as though the RECORD statement had appeared in your program. See Chapter 7 for more information about the RECORD statement. For a thorough understanding of the CDD, you should also be familiar with the *VAX–11 Common Data Dictionary Utilities Reference Manual* and the *VAX–11 Common Data Dictionary Data Definition Language Reference Manual*.

9.2 The VAX–11 Common Data Dictionary (CDD)

When you extract a record definition from the CDD, you can choose whether to include this translated record in the program's listing by using the /SHOW:([NO]CDD_DEFINITIONS) qualifier to the DCL BASIC command, or by using the SET CDD_DEFINITIONS command in the BASIC environment.

Even if you choose not to list the extracted record in BASIC RECORD statement format, the names, data-types, and offsets of the CDD record components are displayed in the program listing's allocation map.

Each time you extract a record from the CDD, an audit or history entry can be logged in the CDD data base. This entry provides a history of the users accessing the CDD. You specify the audit entry to be inserted with the /AUDIT qualifier to the DCL BASIC command, or with the SET AUDIT command in the BASIC environment. The format of the AUDIT qualifier is:

```
            { : str-lit   }
[NO]AUDIT   { : file-spec }
```

where:

str-lit — Is the text to be included in the audit entry. Str-lit must be a quoted string.

file-spec — Is a text file. The audit entry contains up to the first 64 lines of the specified file. The file-spec string cannot be enclosed in quotes.

The default is /NOAUDIT. When you use the /AUDIT qualifier, BASIC also includes the following information in addition to the str-lit or file-spec you specify:

- The access was by way of a BASIC program.
- The access was an extraction (marked in the history log as COMPILE).
- The name of the program module that requested the extraction and the time and date of the request.

The CDD includes only this information if you do not specify an argument to the /AUDIT qualifier. If you specify /NOAUDIT, no history entry for the extraction is made.

You access CDD record definitions with the %INCLUDE compiler directive. When accessing the CDD, the format of %INCLUDE is:

```
%INCLUDE %FROM %CDD cdd-path-name
```

where:

cdd-path-name — Is a quoted string containing the path name of a CDD record node.

There are two types of CDD path names, *absolute* and *relative*. An absolute path name begins with CDD$TOP and specifies the complete path to the record definition. A relative path name begins with any string other than CDD$TOP. When using relative path names, you must have the logical name CDD$DEFAULT defined. For example:

```
$ DEFINE CDD$DEFAULT CDD$TOP.BASIC
```

This logical name definition specifies the beginning of the CDD path name, thus a relative path name specifies the remainder of the path to the record definition. Note also that a CDD path name beginning with CDD$TOP overrides the default CDD path name.

When BASIC translates the CDD definition, each line of the RECORD statement begins with the letter "C" followed by a number. The "C" tells you that the RECORD statement was translated from a CDD record definition. The number tells you whether the CDD extraction itself came from a %INCLUDE file. For example, if your source program directly extracts a CDD record definition, then each line is preceded by a "C1". If the CDD extraction itself came from a %INCLUDE file, then each line of the record definition is preceded by a "C2", and so on.

The following example shows a record defined by the Data Definition Language Utility (CDDL) and the corresponding BASIC RECORD statement.

CDD Definition

```
DEFINE RECORD basicdef

        DESCRIPTION IS
                /* This is an example record containing */
                /* data-types native to VAX-11 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 NUMERIC
                                SIZE IS 4 DIGITS.

                        old     DATATYPE IS PACKED NUMERIC
                                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

```
       1  1 %INCLUDE %FROM %CDD 'BASICDEF'
C1        1        !   This is an example record containing
C1        1        !   data-types native to VAX-11 BASIC
C1        1        RECORD  EMPLOYEE                        ! UNSPECIFIED
C1        1          STRING  STREET   = 30                 ! TEXT
C1        1          STRING  CITY   = 30                   ! TEXT
C1        1          STRING  STATE   = 2                   ! TEXT
C1        1          GROUP   ZIP_CODE                      ! UNSPECIFIED
C1        1            DECIMAL(4 ,0 ) NEW                  ! PACKED NUMERIC
C1        1            DECIMAL(5 ,0 ) OLD                  ! PACKED NUMERIC
C1        1          END GROUP
C1        1          WORD    EMP_NUMBER                    ! SIGNED WORD
C1        1          STRING  WAGE_CLASS   = 2              ! TEXT
C1        1          DOUBLE  SALARY_YTD                    ! D_FLOATING
C1        1        END RECORD
```

This RECORD statement is a translation of the CDD record definition specified by the CDD path name, BASICDEF.

BASIC includes as comment fields the explanatory text in the CDDL DESCRIPTION IS clause. BASIC adds explanatory comment fields to certain lines of the RECORD statement as well. These comments tell you the data type of each elementary RECORD component. In BASIC, neither a GROUP nor a RECORD can have a data type; the BASIC RECORD statement always has the comment "UNSPECIFIED" after each RECORD or GROUP.

The RECORD name corresponds to the field name specified in the first CDDL STRUCTURE statement. BASIC translates any subsequent CDDL STRUCTURE statement to a BASIC GROUP statement. The elementary RECORD components correspond to the subordinate field names within the CDD STRUCTURE statements.

The previous example uses only data types supported both by the CDD and BASIC. However, the CDD supports many data types that are not native to BASIC. Table 9–1 shows the CDD data types that are directly translated to BASIC data types.

Table 9–1: Data Types Common to the CDD and BASIC

CDD Data Type	BASIC Data Type
TEXT	STRING
SIGNED BYTE	BYTE
SIGNED WORD	WORD
SIGNED LONGWORD	LONG
F_FLOATING	SINGLE
D_FLOATING	DOUBLE
G_FLOATING	GFLOAT
H_FLOATING	HFLOAT
PACKED NUMERIC	DECIMAL

If you access CDD definitions that contain data types not supported by BASIC, the compiler reports the warning "CDDSUBGRO, GROUP substituted for datatype" and creates a BASIC GROUP to contain the field. BASIC uses the CDD field name to name the GROUP and creates names for the record components within the GROUP. When creating these names, BASIC uses the BASIC data type (for example, STRING) that it substituted for the CDD data type, followed by "_VALUE". Thus if BASIC substitutes a GROUP containing a string for a CDD data type like LEFT SEPARATE NUMERIC, the record component name would be STRING_VALUE. Table 9–2 shows the CDD data types that do not have a corresponding BASIC data type, and the translation made by BASIC.

Note that the scheme for name creation is slightly more complicated when dealing with complex numbers. See Section 9.1.3 for more information about complex numbers.

Table 9–2: CDD Data Types and BASIC Translation

CDD Data Type	BASIC Translation
UNSIGNED BYTE	GROUP cdd-field-name BYTE END GROUP
UNSIGNED WORD	GROUP cdd-field-name WORD END GROUP

Table 9–2: CDD Data Types and BASIC Translation (Cont.)

CDD Data Type	BASIC Translation
UNSIGNED LONGWORD	GROUP cdd-field-name LONG END GROUP
SIGNED QUADWORD	GROUP cdd-field-name STRING STRING_VALUE = 8 END GROUP
UNSIGNED QUADWORD	GROUP cdd-field-name STRING STRING_VALUE = 8 END GROUP
SIGNED OCTAWORD	GROUP cdd-field-name STRING STRING_VALUE = 16 END GROUP
UNSIGNED OCTAWORD	GROUP cdd-field-name STRING STRING_VALUE = 16 END GROUP
UNSIGNED QUADWORD	GROUP cdd-field-name STRING STRING_VALUE = 8 END GROUP

F_FLOATING COMPLEX	GROUP cdd-field-name
	    SINGLE SINGLE_R_VALUE
	    SINGLE SINGLE_I_VALUE
	END GROUP

D_FLOATING COMPLEX	GROUP cdd-field-name
	    DOUBLE DOUBLE_R_VALUE
	    DOUBLE DOUBLE_I_VALUE
	END GROUP

G_FLOATING COMPLEX	GROUP cdd-field-name
	    GFLOAT GFLOAT_R_VALUE
	    GFLOAT GFLOAT_I_VALUE
	END GROUP

H_FLOATING COMPLEX	GROUP cdd-field-name
	    HFLOAT HFLOAT_R_VALUE
	    HFLOAT HFLOAT_I_VALUE
	END GROUP

SIGNED NUMERIC	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
UNSIGNED NUMERIC	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
LEFT SEPARATE NUMERIC	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
LEFT OVERPUNCHED NUMERIC	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP

(continued on next page)

Table 9–2: CDD Data Types and BASIC Translation (Cont.)

CDD Data Type	BASIC Translation
RIGHT SEPARATE NUMERIC	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
RIGHT OVERPUNCHED NUMERIC	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
VARYING STRING	GROUP cdd-field-name WORD WORD_VALUE STRING STRING_VALUE = length END GROUP
BIT	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
DATE	GROUP cdd-field-name STRING STRING_VALUE = length END GROUP
VIRTUAL FIELD	Not Supported

The following sections describe the translation of CDD data-types.

9.2.1 Character String Data Types

There are two types of CDD character string data type: 1) TEXT and 2) VARYING STRING. The TEXT data type is translated directly into the BASIC STRING data type. Because VARYING STRING is not a BASIC data type, BASIC creates a GROUP to contain the field. For example, here is a CDDL file defining these types of character strings:

```
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.
```

This is the translated RECORD statement, as it appears in the program listing:

```
C1      1       !   test
C1      1       RECORD  BASICSTRINGS                      ! UNSPECIFIED
C1      1         STRING  ABC  = 10                       ! TEXT
C1      1         GROUP   XYZ                             ! VARYING STRING
C1      1           WORD WORD_VALUE
C1      1           STRING  STRING_VALUE  = 16
C1      1         END GROUP
C1      1       END RECORD
```

In the VARYING STRING data type, the actual character string is preceded by a 16–bit count field. Therefore, BASIC creates a WORD variable to hold the specified string length.

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 32767 is interpreted by BASIC as a negative number.

In the preceding example, the BASIC GROUP name (XYZ) is the same as the name of the field in the CDDL file. Therefore, BASIC must supply an additional name for the RECORD components. The supplied names are WORD_VALUE and STRING_VALUE. For example, this program fragment creates an instance of the RECORD named BASICSTRINGS:

```
100       MAP (TEST) BASICSTRINGS MY_REC
```

The names you use to reference these components are MY_REC::XYZ::WORD_VALUE and MY_REC::XYZ::STRING_VALUE.

9.2.2 Integer (Fixed-Point) Data Types

The CDD supports 8–bit (BYTE), 16–bit (WORD), 32–bit (LONGWORD), 64–bit (QUADWORD), and 128–bit (OCTAWORD) integer data types. Each of these data types can have the following additional attributes:

- SIGNED
- UNSIGNED
- SIZE
- DIGITS
- FRACTION
- BASE

BASIC directly supports only signed BYTE, signed WORD, and signed LONGWORD integers. If the CDD definition describes a BYTE, WORD, or LONGWORD integer as UNSIGNED, BASIC signals the diagnostic warning error "CDDSUBGRO, GROUP substituted for datatype" and places the field in a BASIC GROUP. The GROUP name is the same as the CDD field name and BASIC assigns a new name to the field. For example:

CDD 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 unsigned byte.

        my_word          datatype is signed word.

        my_uword         datatype is unsigned word.
```

(continued on next page)

```
        my_long            datatype is signed longword.

        my_ulong           datatype is unsigned longword.

    end basicint structure.
end integers.
```

Translated RECORD Statement

```
      1 1 %include %from %cdd 'integers'
C1    1          !   Test of selected integer data-types
C1    1          RECORD  BASICINT                       ! UNSPECIFIED
C1    1            BYTE     MY_BYTE                     ! SIGNED BYTE
C1    1            GROUP    MY_UBYTE                    ! UNSIGNED BYTE
C1    1              BYTE     BYTE_VALUE
C1    1            END GROUP
C1    1            WORD     MY_WORD                     ! SIGNED WORD
C1    1            GROUP    MY_UWORD                    ! UNSIGNED WORD
C1    1              WORD     WORD_VALUE
C1    1            END GROUP
C1    1            LONG     MY_LONG                     ! SIGNED LONGWORD
C1    1            GROUP    MY_ULONG                    ! UNSIGNED LONGWORD
C1    1              LONG     LONG_VALUE
C1    1            END GROUP
C1    1          END RECORD
```

When this record definition is extracted from the CDD, BASIC signals "CDDSUBGRO, GROUP substituted for datatype" for each of the unsigned data types. Note that the name of each GROUP is the same as the name for the CDD field. BASIC names the unsigned byte BYTE_VALUE, the unsigned word WORD_VALUE, and the unsigned longword LONG_VALUE.

BASIC does not directly support QUADWORD or OCTAWORD integers. If a CDD definition contains such an integer field, BASIC creates a GROUP with the same name as the integer field, and creates a string component within the group. This string is 8 bytes for QUADWORD integers and 16 bytes for OCTAWORD integers. For example:

CDD 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 1 %include %from %cdd 'bigintegers'
C1    1          !   Test of quadword and octaword integer data-types
C1    1          RECORD  BASICINT                       ! UNSPECIFIED
C1    1            GROUP    MY_QUAD                     ! SIGNED QUADWORD
C1    1              STRING   STRING_VALUE  = 8
C1    1            END GROUP
C1    1            GROUP    MY_OCTA                     ! SIGNED OCTAWORD
C1    1              STRING   STRING_VALUE  = 16
C1    1            END GROUP
C1    1          END RECORD
```

BASIC signals the warning error "CDDSUBGRO, GROUP substituted for datatype" for each occurrence of a QUADWORD or OCTAWORD CDD field. For unsigned QUADWORD and OCTAWORD fields, BASIC inserts the comment "! UNSIGNED QUADWORD" or "! UNSIGNED OCTAWORD".

The CDD refers to integer data types as *fixed-point* data types because they can have attributes that BASIC integers cannot. For example, in the CDD definition, you can specify that fixed-point data types have an implied exponent with the SCALE keyword. You can also specify that SCALE for fixed-point fields is to be interpreted in a numeric base other than 10 with the BASE keyword. Because BASIC integers do not have these attributes, BASIC reports the warning error "CDDATTSCA, CDD specifies SCALE for <name>. Not supported" for fixed-point fields containing a SCALE specification. Similarly, BASIC reports "CDDATTBAS, CDD attributes for <name> are other than base 10" for fixed-point fields specifying a base other than 10. For example:

CDD 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.

        my_quad          datatype is signed quadword scale 5.

        my_octa          datatype is signed octaword base 16.

    end basicint structure.
end funnyintegers.
```

Translated RECORD Statement

```
        1  1 %include %from %cdd 'funnyintegers'
C1      1          !   Test of quadword and octaword integer data-types
C1      1          RECORD  BASICINT                          ! UNSPECIFIED
C1      1            BYTE     MY_BYTE                        ! SIGNED BYTE
C1      1            LONG     MY_LONG                        ! SIGNED LONGWORD
C1      1            GROUP    MY_QUAD                        ! SIGNED QUADWORD
C1      1              STRING   STRING_VALUE   = 8
C1      1            END GROUP
C1      1            GROUP    MY_OCTA                        ! SIGNED OCTAWORD
C1      1              STRING   STRING_VALUE   = 16
C1      1            END GROUP
C1      1          END RECORD
```

Note that, at compile time, BASIC also reports these warning errors for each reference to fields that are not base 10 or that have a SCALE.

9.2.3 Floating-Point Data Types

The CDD supports F_floating, D_floating, G_floating, and H_floating data types. These correspond to the BASIC LE (F_floating), DOUBLE (D_floating), GFLOAT (G_floating), and HFLOAT (H_floating) data types. As with fixed-point data types, the CDD also allows the specification of SCALE and BASE for floating-point numbers. If a CDD field contains a floating-point field that specifies SCALE or BASE,

BASIC reports the warning errors "CDDATTSCA, CDD specifies SCALE for <name>. Not supported" and "CDDATTBAS, CDD attributes for <name> are other than base 10". For example:

CDD Definition

```
define record cdd$top.basic.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.

        my_hfloat           datatype is h_floating.

    end basicfloat structure.
end floats.
```

Translated RECORD Statement

```
       1  1 %include %from %cdd 'floats'
C1     1           !   Test of floating-point data-types
C1     1           RECORD  BASICFLOAT                      ! UNSPECIFIED
C1     1             SINGLE   MY_SINGLE                    ! F_FLOATING
C1     1             DOUBLE   MY_DOUBLE                    ! D_FLOATING
C1     1             GFLOAT   MY_GFLOAT                    ! G_FLOATING
C1     1             HFLOAT   MY_HFLOAT                    ! H_FLOATING
C1     1           END RECORD
```

In addition, the CDD supports complex floating-point numbers. Complex numbers consist of a real and an imaginary part. Each part requires the same amount of storage as a simple floating-point number. Thus, each floating-point complex number requires twice as much storage as a simple floating-point number.

When extracting a CDD record that contains complex numbers, BASIC creates a GROUP to contain the field, names the GROUP with the same name as the CDD field, and reports the warning error "data type in CDD not supported, substituting GROUP". BASIC also assigns names to real and imaginary parts of the complex number. As before, BASIC uses the data-type and "_VALUE" to create the name, but because each complex number contains both a real and an imaginary part, BASIC adds an "_R" to the name of the real part and an "_I" to the name of the imaginary part. For example:

CDD 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.

        my_h_complex_1   datatype          h_floating_complex.

    end complex structure.
end complex.
```

Translated RECORD Statement

```
      1  1 %include %from %cdd 'complex'
C1    1          !    test complex data-types
C1    1          RECORD  COMPLEX                        ! UNSPECIFIED
C1    1            GROUP   MY_S_COMPLEX_1               ! F_FLOATING_COMPLEX
C1    1              SINGLE  SINGLE_R_VALUE
C1    1              SINGLE  SINGLE_I_VALUE
C1    1            END GROUP
C1    1            GROUP   MY_D_COMPLEX_1               ! D_FLOATING_COMPLEX
C1    1              DOUBLE  DOUBLE_R_VALUE
C1    1              DOUBLE  DOUBLE_I_VALUE
C1    1            END GROUP
C1    1            GROUP   MY_G_COMPLEX_1               ! G_FLOATING_COMPLEX
C1    1              GFLOAT  GFLOAT_R_VALUE
C1    1              GFLOAT  GFLOAT_I_VALUE
C1    1            END GROUP
C1    1            GROUP   MY_H_COMPLEX_1               ! H_FLOATING_COMPLEX
C1    1              HFLOAT  HFLOAT_R_VALUE
C1    1              HFLOAT  HFLOAT_I_VALUE
C1    1            END GROUP
C1    1          END RECORD
```

9.2.4 Decimal String Data Types

The CDD supports these forms of decimal numeric string:

- UNSIGNED NUMERIC
- LEFT SEPARATE NUMERIC
- LEFT OVERPUNCHED NUMERIC
- RIGHT SEPARATE NUMERIC
- RIGHT OVERPUNCHED NUMERIC
- SIGNED NUMERIC
- PACKED NUMERIC

The only one of these directly supported by BASIC is PACKED NUMERIC, which corresponds to the BASIC DECIMAL data type. For all other decimal string data types, BASIC places the field in a GROUP with the same name as the CDD field and creates a string record component to contain the field. For example:

CDD Definition

```
define record cdd$top.basic.decimalstring
    description is
        /* test decimal string data-types */.

    decimalstring structure.

        my_packed_numeric        datatype is packed numeric size is
                                         5 digits 2 fractions.

        my_unsigned_numeric      datatype is unsigned numeric size is
                                         8 digits 4 fractions.

        my_s_num_lef_sep         datatype is signed numeric left separate
                                         size is 10 digits 3 fractions.
```

(continued on next page)

```
        my_s_num_left_ovpnch       datatype is signed numeric left overpunched
                                            size is 5 digits 2 fractions.

        my_s_num_right_sep         datatype is signed numeric right separate
                                            size is 3 digits 1 fractions.

        my_s_num_right_ovpnch      datatype is signed numeric right overpunched
                                            size is 4 digits 2 fractions.

    end decimalstring structure.
end decimalstring.
```

Translated RECORD Statement

```
        1  1 %include %from %cdd 'decimalstring'
C1      1          !    test decimal string data-types
C1      1          RECORD  DECIMALSTRING                 ! UNSPECIFIED
C1      1            DECIMAL(5 ,2 ) MY_PACKED_NUMERIC    ! PACKED NUMERIC
C1      1            GROUP    MY_SIGNED_NUMERIC          ! SIGNED NUMERIC
C1      1              STRING  STRING_VALUE  = 6
C1      1            END GROUP
C1      1            GROUP    MY_UNSIGNED_NUMERIC        ! UNSIGNED NUMERIC
C1      1              STRING  STRING_VALUE  = 8
C1      1            END GROUP
C1      1            GROUP    MY_S_NUM_LEF_SEP           ! NUMERIC LEFT SEPARATE
C1      1              STRING  STRING_VALUE  = 11
C1      1            END GROUP
C1      1            GROUP    MY_S_NUM_LEFT_OVPNCH       ! NUMERIC LEFT OVERPUNCHED
C1      1              STRING  STRING_VALUE  = 5
C1      1            END GROUP
C1      1            GROUP    MY_S_NUM_RIGHT_SEP         ! NUMERIC RIGHT SEPARATE
C1      1              STRING  STRING_VALUE  = 4
C1      1            END GROUP
C1      1            GROUP    MY_S_NUM_RIGHT_OVPNCH      ! NUMERIC RIGHT OVERPUNCHED
C1      1              STRING  STRING_VALUE  = 4
C1      1            END GROUP
C1      1          END RECORD
```

9.2.5 Other Data Types

The CDD supports the following data types that are not supported by BASIC:

- BIT
- DATE
- VIRTUAL

The BIT data type specifies that the field is a string of bits. The DATE data type describes a 64–bit VAX–11 absolute date structure. The VIRTUAL data type describes a VAX–11 DATATRIEVE virtual field.

You can extract CDD definitions containing BIT fields; however, the field must be a multiple of eight bits (one byte). This means that the following field must be aligned on a byte boundary. BASIC signals the informational error "CDDSUBGRO, data type in CDD not supported, substituted group for <field name>" for any BIT field. If the following field is not aligned on a byte boundary, BASIC signals the fatal error "CDDBITFLD, field <name> from CDD has bit offset or length".

BASIC does not support VIRTUAL fields. An attempt to extract a CDD definition containing such a field causes BASIC to signal the fatal error "CDDUNSDAT, data type specified in CDD for <name> not supported".

BASIC does support DATE fields by placing them in a GROUP and assigning a length of eight bytes to a STRING_VALUE variable in the GROUP.

9.2.6 Arrays

The CDD supports three types of arrays:

- Multidimensional arrays (ARRAY clause)
- One-dimensional, fixed length arrays (OCCURS clause)
- One-dimensional, variable length arrays (OCCURS DEPENDING clause)

Arrays are valid for any CDD data type. However, BASIC does not support arrays of the entire CDD definition. That is, the translated RECORD statement cannot itself be an array.

Variable length arrays in the CDD must have a minimum and maximum number of elements specified. BASIC always uses the maximum size when translating to a RECORD. For example:

CDD Definition

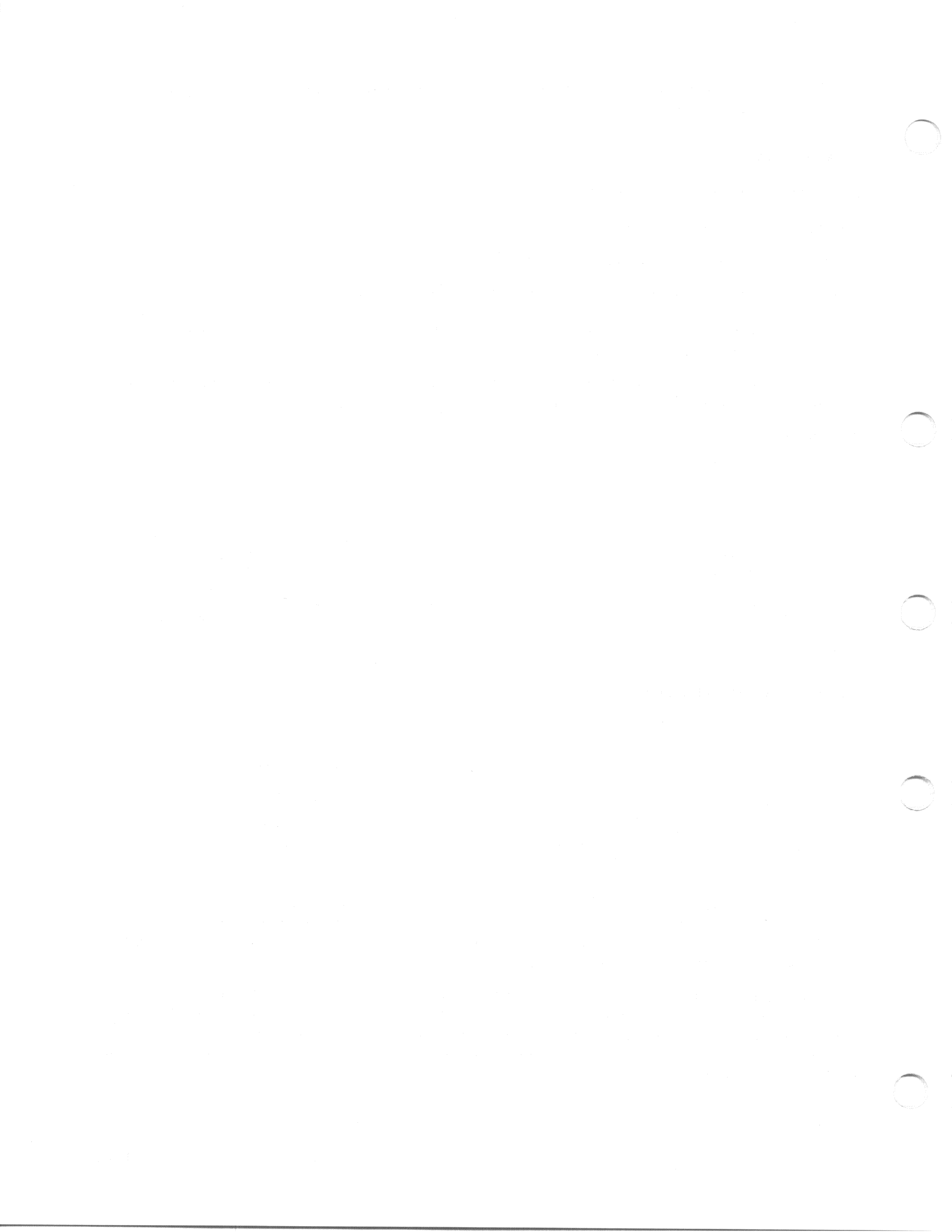
```
define record cdd$top.basic.array1
    description is
        /* test arrays */.
    array_1 structure.
        my_byte         longword        datatype        signed byte.
        my_string       array 0:10              datatype        text size 10.
        my_s_real       array 0:2 0:4           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 2 to 4 times depending my_byte
                                                datatype        h_floating.
    end array_1 structure.
end array1.
```

Translated RECORD Statement

```
        1 1 %include %from %cdd 'array1'
C1      1           !    test arrays
C1      1           RECORD  ARRAY_1                     ! UNSPECIFIED
C1      1             BYTE     MY_BYTE                  ! SIGNED BYTE
C1      1             STRING   MY_STRING(10) = 10       ! TEXT
C1      1             SINGLE   MY_S_REAL(2,4)           ! F_FLOATING
C1      1             DOUBLE   MY_D_REAL(2)             ! D_FLOATING
C1      1             GFLOAT   MY_G_REAL(3)             ! G_FLOATING
C1      1             HFLOAT   MY_H_REAL(3)             ! H_FLOATING
C1      1           END RECORD
```

Note that if an array in the CDD definition is not zero-based, BASIC adjusts the bounds so that the translated RECORD statement specifies a zero-based array. For example, the array named MY_D_REAL has bounds of 1:3 in the CDD definition, but BASIC translates this to an array with a lower bound of 0 and an upper bound of 2.

By default, arrays in the CDD are row-major. This means that when storage is allocated for the array, the rightmost subscript varies fastest. All BASIC arrays are row-major. Although the CDD also allows you to specify column-major arrays, BASIC does not support them. If a CDD definition containing a column-major array is extracted, BASIC signals the fatal error "CDDCOLMAJ, <array-name> from CDD is a column major array".



Chapter 10
Object Module Libraries and Shareable Images

Object module libraries are files containing object modules and related information, including a list of the names of the modules and a list of the global symbols contained in them. Libraries contain commonly used routines that can be linked to your programs.

Shareable images are similar to object 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 libraries or shareable images lets you access them at link time. This means that you need not include the routines in the source code, thus increasing efficiency at compile 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. This is because one copy of the executable code is shared by all users.

For a thorough understanding of libraries and shareable images, you should refer to the *VAX–11 Linker Reference Manual* and the *VAX–11 Guide to Creating Modular Library Procedures*. See the *VAX–11 Utilities Reference Manual* for more information on installing shareable images.

10.1 User Supplied Libraries

You can access object module libraries from both the BASIC environment and DCL command level. When linking programs at DCL command level, libraries can contain object code created by any VAX–11 native mode compiler or assembler. However, when you RUN a program in the BASIC environment, any libraries accessed can contain only VAX–11 BASIC object code, that is, object code from SUB and FUNCTION subprograms.

10.1.1 Creating Libraries

You create an object module library with the LIBRARY DCL command. Its format is:

```
LIBRARY/CREATE library-file-spec [input-file-spec[, . . .]]
```

where:

library-file-spec	Is the file specification for the library you are creating.
[input-file-spec[,...]]	Is a list of object modules to be inserted in the library.

For example:

```
$ BASIC MODULE1,MODULE2
$ LIBRARY/CREATE TESTLIB1.OLB MODULE1.OBJ,MODULE2.OBJ
```

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 it. See the *VAX/VMS Command Language User's Guide* for more information on the LIBRARY command.

10.1.2 Accessing User Supplied Libraries

User supplied libraries can be accessed by specifying the /LIBRARY qualifier to the DCL LINK command. For example:

```
$ LINK MAIN,TESTLIB/LIBRARY
```

This command causes the linker to search TESTLIB.OLB if there are unresolved symbols in the BASIC object module MAIN.OBJ. You can also explicitly include a module from a library with the /INCLUDE qualifier:

```
$ LINK MAIN,TESTLIB/LIBRARY/INCLUDE=MODULE1,MODULE2
```

This command causes the linker to include MODULE1 and MODULE2 from TESTLIB.OLB, whether or not it needs these modules to resolve symbols.

You can also access your own libraries automatically. To automatically access object module libraries in the BASIC environment, you must assign them as logical names of the form:

BASIC$LIBn

where:

n Is a number from zero to nine, inclusive.

For example:

```
$ ASSIGN DBA0:[SMITH]TESTLIB.OLB BASIC$LIB0
```

After you enter this command, a program executing in the BASIC environment automatically accesses DBA0:[SMITH]TESTLIB.OLB to resolve program symbols. Note that this command assigns BASIC$LIB0 as a process-wide logical name. A privileged user can also assign a BASIC library as a group- or system-wide logical name.

The VAX–11 Linker does not automatically search user libraries unless they have been assigned as logical names of the form:

```
LNK$LIBRARY and LNK$LIBRARY_1 through LNK$LIBRARY_999
```

This means that when you are linking programs at DCL command level, the VAX–11 Linker does not automatically search libraries that have been assigned as BASIC$LIB logical names. You must assign the library files as LNK$LIBRARY logical names. Also, if you have more than one library for the linker to search, you must assign the first one as LNK$LIBRARY, the second one as LNK$LIBRARY_1, the third as LNK$LIBRARY_2, and so on. If you do not number these libraries consecutively, the linker does not search past the first missing logical name.

For example, to make your user libraries known to the linker, assign them as LNK$LIBRARY logical names as follows:

```
$ ASSIGN DBA0:[SMITH]TESTLIB.OLB LNK$LIBRARY
$ ASSIGN DBA0:[SMITH]TESTLIB1.OLB LNK$LIBRARY_1
$ ASSIGN DBA0:[SMITH]TESTLIB2.OLB LNK$LIBRARY_2
```

10.2 The System Library

If symbols are unresolved after the linker searches all input modules and user-specified libraries, the linker goes on to search the files in the default system library. The VAX/VMS system supplies a set of system libraries:

- A shareable image symbol table library (IMAGELIB.OLB)
- The default system object module library (STARLET.OLB)
- The default system macro library (STARLET.MLB)

IMAGELIB.OLB contains the symbol tables for the parts of the VAX–11 Common Run-Time Library (RTL) that are in shareable images.

If the linker needs to search the default system library, it searches the shareable image symbol table library (IMAGELIB.OLB) first. If program symbols remain unresolved, the linker searches STARLET.OLB.

STARLET.OLB is an object module library containing the object files used to create the shareable image version of the RTL, as well as other less frequently used procedures. This object library also contains modules for interfacing to VAX/VMS System Services.

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. VMSRTL.EXE, SCRSHR.EXE, LBRSHR.EXE and CRFSHR.EXE
4. STARLET.OLB

If the linker finds no needed routines in the RTL shareable images, it does not include any 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.

Note that the linker searches user supplied libraries before searching the default system library. This means that, if one of your modules has the same name (program symbol) as a VAX–11 System Service or an RTL routine, the linker will include your module in the resulting image rather than the system service or RTL routine.

10.3 Shareable Images

Shareable images contain executable code that can be shared by other images. Shareable images are not directly executable; they are intended to be included (by the linker) in other images.

Shareable images provide benefits by:

- Conserving disk storage space
- Conserving main physical memory
- Reducing paging I/O
- Allowing shared memory-resident data bases
- Eliminating the need to relink programs that access a new version of a shared routine

Note that some of these benefits can be realized only if the shareable image is installed using the Install Utility (INSTALL).

When an executable image is linked with a shareable image, the contents of the shareable image are normally not copied into the executable image file. Therefore, you need only one copy of the shareable image on disk. However, if the shareable image has not been installed, each program linked against the shareable image gets its own copy of the shareable image at run time.

If a shareable image has been installed, you conserve physical memory and reduce paging I/O. In this case, many processes can include the physical memory pages of an installed shareable image in their address space, thus reducing the requirements for physical memory.

When a program is linked with a shareable image, the required shareable image code is not included in the created executable image on disk. This code is included by the image activator at run time. Thus, many programs can reside on disk and be bound with a particular shareable image, and only one physical copy of that shareable image file need exist on disk.

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. Thus, if a page fault occurs for a shared page, the shared page may already be resident in memory and in this case, no paging I/O is required.

To create a shareable image, use the /SHAREABLE qualifier to the DCL LINK command:

```
$ LINK/SHAREABLE prog1[,prog2...]
```

where:

prog1 and prog2 Are object modules created by BASIC.

This command creates an image that can be linked to other programs. You cannot execute a shareable image with the DCL RUN command.

To link a program with an existing shareable image, you must use an options file to specify that the shareable image is input to the linker. The options file should contain the name of the shareable image followed by the /SHAREABLE qualifier.

After a program is linked with a shareable image, the image activator locates the shareable image when the program is executed. The default directory for shareable images is SYS$SHARE

(SYS$SYSROOT:[SYSLIB]). If you want the image activator to access a shareable image in a directory other than SYS$SHARE:, you must define a logical name for the shareable image. That is, you must assign the full file specification of the shareable image to the name of the shareable image, as follows:

```
$ DEFINE MYSHR DISK$WORKDISK:[MYDIR]MYSHR
```

For example, this is the program to be inserted in a shareable image:

ADD.BAS

```
10          FUNCTION REAL ADD (LONG A, LONG B)
            ADD = A + B
            FUNCTIONEND
```

You compile this program and link it with this command:

```
$ LINK ADD/SHAREABLE,ADDSUB/OPTION
```

This command creates a shareable image (ADD.EXE) from the object file (ADD.OBJ). This is the options file required for the link operation:

ADDSUB.OPT

```
UNIVERSAL = ADD
```

This option file ensures that the ADD program is available to both the linker and the image activator.

This is the program that accesses the ADD routine in the shareable image:

CALLADD.BAS

```
10          EXTERNAL REAL FUNCTION ADD (LONG, LONG)
            DECLARE LONG X,Y
            X = 1
            Y = 2
            PRINT ADD(X,Y)
200         END
```

Once this program has been compiled, you link it with the shareable image ADD.EXE:

```
$ LINK CALLADD,ADDMAIN/OPTION
```

This requires a linker options file specifying that ADD is a shareable image:

ADDMAIN.OPT

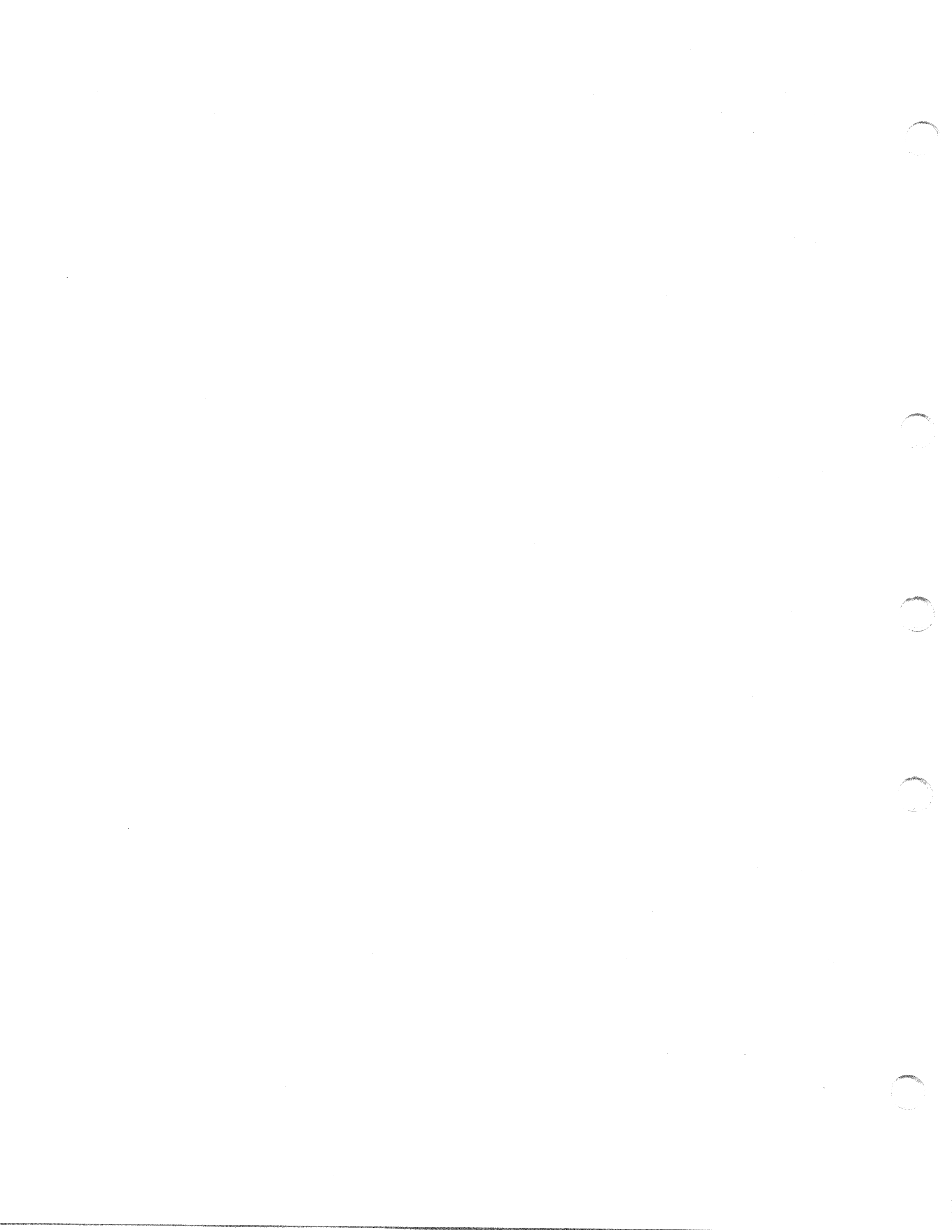
```
WRK$$DISK:[SMITH.BASIC]ADD/SHAREABLE=NOCOPY
```

Now a logical name must be defined for the shareable image so that the image activator finds the shareable image in WRK$$DISK:[SMITH.BASIC] rather than in the default library for shareable images (SYS$SHARE:):

```
$ ASSIGN WRK$$DISK:[SMITH.BASIC]ADD.EXE ADD
```

The CALLADD program can now be executed.

Note that this has been a very simple example of using shareable images. For a thorough understanding of shareable images, see the *VAX–11 Linker Reference Manual*.



Appendix 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

BASIC diagnoses compile-time errors and:

- 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, the statement number within the line, the location number as previously displayed, and the message text. This is repeated for each error in the line.

BASIC repeats this procedure for each error diagnosed during compilation. The error message format for compile-time errors is:

%BASIC–<l>–<mnemonic>, S <n>, <n>: <message>

where:

<l> Is a letter indicating the severity of the error. The severity indicator can be:

- 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.

S <n> Is the nth statement within the displayed line (the statement containing the error).

<n>: Is the nth error within the line's "picture."

<message> Is the text of the error message.

For example:

```
Error on line 10

        10 DECLARE REAL BYTE A, A
........................1.......2


%BASIC-E-CONDATSPC, S 1, 1:   conflicting data type specifications
%BASIC-E-ILLMULDEF, S 1, 2:   illegal multiple definition of name A
```

This display tells you that two errors were detected on line 10; 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 statement in the line contained the error (the first statement)
- 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.

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.

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.

APPSCRINC, APPEND/SCRATCH access inconsistent with file organization

Explanation: ERROR – An OPEN statement specifies ACCESS APPEND or SCRATCH for a relative or indexed file.

User Action: Change the ACCESS clause; ACCESS APPEND or SCRATCH is valid only for sequential files.

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; you cannot pass an entire array as a parameter to a DEF function.

ARRTOOBIG, named array <array-name> is too large

Explanation: ERROR – An array requires more than $(2^{29} - 1)$ bytes of storage.

User Action: Reduce the size of the array.

ATROVRVAR, attributes of overlaid variable <name> don't match

Explanation: WARNING – 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.

BASICHLB, BASIC's HELP library is not installed on this system

Explanation: ERROR – A HELP command was entered and the BASIC HELP library was not available.

User Action: See your system manager.

BASICSHR, failure in loading SYS$LIBRARY:BASICSHR.EXE

Explanation: ERROR – The RUN command or an immediate mode statement was entered and the compiler could not find the executable image containing the code that supports these operations.

User Action: See your system manager.

BIFREQNUM, numeric expression is needed in built-in function

Explanation: ERROR – A reference to a BASIC built-in function contains a string instead of a numeric expression.

User Action: Supply a numeric expression.

BIFREQSTR, string expression is needed in built in function

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 VAX–11 BASIC.

User Action: Remove the function reference.

BOUCANNOT, bound cannot be specified for array <array-name>

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 <array-name>

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)
```

CDDACCERR, CDD access error

Explanation: ERROR – The CDD detected an error on an attempted CDD record extraction. BASIC displays the CDD error.

User Action: Take action based on the associated CDD error.

CDDACCREC, CDD error while accessing record

Explanation: ERROR – The CDD reported an error when accessing the record. The CDD record definition is corrupt or there is an internal error in either BASIC or the CDD.

User Action: If the problem is not in the CDD definition, please submit an 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: INFORMATIONAL – The CDD contains an array field with a lower bound that is not zero. BASIC adjusts the bound so that the array is zero based.

User Action: None.

CDDALCOFF, CDD inconsistent with allocated offset for <field-name>

Explanation: ERROR – The offset of a field within a BASIC RECORD differs from the offset specified by the CDD for that record.

User Action: Please submit an SPR with the source code of a small program that produces this error.

CDDALCSIZ, CDD inconsistent with allocated size for <field-name>

Explanation: ERROR – The amount of storage allocated for a field in a BASIC RECORD differs from the amount specified by the CDD for that record.

User Action: Please submit an SPR with the source code of a small program that produces this error.

CDDALCSPN, CDD inconsistent with allocated span for <field-name>

Explanation: ERROR – The amount of storage allocated by a BASIC RECORD for an array differs from the amount specified by the CDD for that record.

User Action: Please submit an SPR with the source code of a small program that produces this error.

CDDAMBFLD, ambiguous field name <name> for <RECORD-name>

Explanation: ERROR – More than one CDD structure share the same level and the same name.

User Action: Change the CDD definition so that the structures have different names.

CDDATTBAS, CDD attributes for <name> are other than base 10

Explanation: ERROR – A field in a CDD 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 the CDD or avoid using the field.

CDDATTDAT, CDD data type attribute not permitted for GROUP

Explanation: ERROR – A CDD definition specified a data type after the CDD STRUCTURE keyword. BASIC translates STRUCTURE to a BASIC RECORD or GROUP statement. These BASIC statements do not allow data-type attributes.

User Action: Change the CDD definition.

CDDATTDIG, DIGITS attribute of <field-name> not supported for datatype

Explanation: INFORMATIONAL – The field contains a CDD fixed-point data type that specifies the number of allowed digits. This warning tells you that BASIC interprets the field as BYTE, WORD, or LONG and does not support the DIGITS attribute for this data type.

User Action: None.

CDDATTITE, CDD error while accessing item <field-name> of record

Explanation: ERROR – The CDD reported an error when accessing the field. The CDD record definition is corrupt, or there is an internal error in either BASIC or the CDD.

User Action: If the problem is not in the CDD definition, please submit an SPR with the source code of a small program that produces this error.

CDDATTSCA, CDD specifies SCALE for <RECORD–component>. Not supported.

Explanation: INFORMATIONAL – A field in a CDD definition uses the SCALE keyword. This warns you that the field has an implied exponent.

User Action: Remove the SCALE attribute in the CDD, or avoid using the field.

CDDBASNAM, CDD specified BASIC name <name> has illegal form

Explanation: ERROR – The BASIC name specified in the CDD record definition is a reserved keyword or contains an illegal character.

User Action: Change the invalid field name.

CDDBITFLD, field <field-name> from CDD has bit offset or length

Explanation: ERROR – A CDD field does not start on a byte boundary.

User Action: Use the CDD ALIGN keyword to align the field following <field-name> on a byte boundary.

CDDCOLMAJ, <array-name> from CDD is a column major array

Explanation: ERROR – An array specified in a CDD definition is column-major rather than row-major. Thus it is incompatible with BASIC arrays.

User Action: None. You cannot extract a CDD definition containing a column-major array.

CDDDIGERR, decimal digits of <value> in CDD out of range for <field-name>

Explanation: ERROR – A packed numeric CDD specifies more than 31 digits.

User Action: Reduce the number of digits specified in the CDD definition.

CDDDIMNOT, RECORD cannot be dimensioned

Explanation: ERROR – A CDD definition is itself an array. This is incompatible with BASIC RECORDs, which can contain arrays but cannot be arrays.

User Action: None. You cannot access CDD definitions that are arrays.

CDDDUPREC, RECORD <name> from CDD has duplicate name

Explanation: ERROR – The CDD record name conflicts with a previous RECORD name. The previous RECORD name may be a standard BASIC RECORD or another CDD record.

User Action: Remove one of the duplicate definitions.

CDDFLDNAM, field name missing

Explanation: ERROR – The CDD definition contains a field that is not named.

User Action: Supply a field name for the CDD definition.

CDDINIIGN, initial value specified in CDD ignored for <field-name>

Explanation: INFORMATIONAL – A CDD field definition specified an initial value. BASIC does not support initial values from the CDD.

User Action: None.

CDDLOWBOU, lower bound omitted for dimension <number> of <array-name>

Explanation: ERROR – An array in a CDD definition does not specify a lower bound.

User Action: Check to make sure the omission is not a mistake. 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 definition specifies more than 32 dimensions.

User Action: Reduce the number of dimensions in the CDD definition.

CDDNAMKEY, <name> is a BASIC keyword

Explanation: ERROR – A CDD definition contains a field name that is a reserved word in BASIC.

User Action: Change the name in the CDD definition or supply a BASIC name clause.

CDDOCCIGN, OCCURS DEPENDING ON clause for <array-name> from CDD ignored

Explanation: INFORMATIONAL – The CDD contains an array field with a variable number of elements. 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 offset error, field <field-name> offsets out of order

Explanation: ERROR – The CDD definition has been corrupted or there is an internal error in either BASIC or the CDD.

User Action: If the problem is not in the CDD definition, please submit an SPR with the source code of a small program that produces this error.

CDDPREERR, decimal precision of <value> in CDD 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 the CDD to be less than or equal to the total number of digits.

CDDRECFOR, CDD record format is not fixed

Explanation: ERROR – The CDD supports both variable and fixed-length records. BASIC supports only fixed-length records.

User Action: Change the CDD record definition to specify fixed-length.

CDDSCAERR, decimal scale of <scale-factor> is out of range for <field> from CDD

Explanation: ERROR – The scale factor for a packed decimal CDD field is greater than the number of digits in the field or less than zero.

User Action: Change the scale factor in the CDD definition.

CDDSCAZER, scale 0 specified for CDD field <field-name>

Explanation: INFORMATIONAL – A CDD field specifies no scale factor for a D_floating field, but the BASIC program specifies a non-zero scale factor.

User Action: Use a scale factor of zero in the BASIC program.

CDDSUBGRO, substituted GROUP for <field-name>. Data type in CDD not supported.

Explanation: INFORMATIONAL – The CDD definition specifies a data type that is not native to BASIC. BASIC creates a GROUP with the same name as the CDD field and creates variable names for the GROUP components.

User Action: None.

CDDTAGIGN, tag value ignored for <field-name> from CDD

Explanation: INFORMATIONAL – The CDD record definition contains a variant field.

User Action: None; however, if you use an alternate field variant, be sure to update the tag field.

CDDATTTXT, CDD TEXT attribute for group <group-name> ignored

Explanation: INFORMATIONAL – A CDD record definition specifies a data type of TEXT for the entire record.

User Action: None. BASIC ignores the TEXT attribute and substitutes the UNSPECIFIED attribute.

CDDUNSDAT, data type specified in CDD for <field-name> not supported

Explanation: ERROR – The data type specified for a field is not supported by BASIC.

User Action: Avoid using this field.

CDDUPPBOU, upper bound omitted for dimension <number> of <array-name>

Explanation: ERROR – An array in a CDD definition does not specify an upper bound.

User Action: Specify an upper bound in the CDD definition.

CDDVARFLD, field <name> from CDD has variable offset or length

Explanation: ERROR – A CDD field can be either variable or fixed-length. BASIC supports only fixed-length fields.

User Action: Either change the CDD definition or avoid using the field.

CHAEXPMUS, channel expression must be numeric

Explanation: ERROR – The program contains a non-numeric 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.

COMMAPALI, variable <name> not aligned in COMMON/MAP <name>

Explanation: WARNING – The total storage preceding a numeric variable in a COMMON or MAP is an odd number of bytes.

User Action: None. BASIC–PLUS–2 pads the preceding storage with a blank byte because the PDP–11 requires that numeric data starts on a word boundary. VAX–11 BASIC does not pad the storage because numeric data can start on any byte boundary. However, if you want the program to run on both types of systems, you should ensure that all numeric data starts on a word boundary.

COMMAPOVF, COM/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.

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.

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.

CORSTAFRA, corrupted stack frame

Explanation: FATAL – An immediate mode statement was entered after a STOP statement was executed in the 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 BASIC environment.

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.

DATTYPEXP, data type required for variable with /EXPLICIT

Explanation: ERROR – A program compiled with the /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 /EXPLICIT qualifier.

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.

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.

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.

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.

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, INPUT A,,B is a

syntax error, but BASIC continues the compilation by ignoring the second 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.

DESOUTRAN, destination out of range

Explanation: ERROR – The branch destination in an ON GOTO or ON GOSUB statement is greater than 32767 bytes away from the statement, or the FOR and NEXT statements in a FOR–NEXT loop have more than 32767 bytes of code between them.

User Action: Reduce the distance between the destination and the statement or reduce the amount of code between the FOR and NEXT statements.

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.

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.

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 /FLAG:OPTIMIZATION 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 or MAP.

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, the terminating line number, or END IF.

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.

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.

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.

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.

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.

EXPTOOCOM, expression too complicated

Explanation: The program contains an expression too complicated to compile.

User Action: Rewrite the expression as two or more less complicated assignment statements.

EXPUNAOPE, expecting unary operator or legal lexical operand

Explanation: FATAL – 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: FATAL – 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.

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.

EXTNAMTOO, EXTERNAL name too long, truncating to <new-name>

Explanation: ERROR – An EXTERNAL statement names a symbol longer than 31 characters for VAX–11 BASIC or 6 characters for PDP–11 BASIC–PLUS–2.

User Action: Shorten the symbol name. External names must be 31 characters or fewer on VAX/VMS systems and 6 characters or fewer on PDP–11 systems.

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. BASIC does not support external string variables.

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.

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

Explanation: FATAL – 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.

FILNOTALL, FILL not allowed in DYNAMIC MAP

Explanation: ERROR – A DYNAMIC MAP statement contains a FILL item.

User Action: Remove the FILL item.

FLDVARNOT, FIELDed variable cannot be a parameter

Explanation: ERROR – The parameter list in a reference to a DEF or a subprogram contains a string variable or string array element that also appears in a FIELD statement.

User Action: Remove the variable or array from the parameter list or the FIELD statement.

FLOCVTILL, floating CVT valid only for SINGLE and DOUBLE

Explanation: ERROR – A CVTF$ or CVT$F function names a GFLOAT or HFLOAT value as an argument.

User Action: Use a SINGLE or DOUBLE argument rather than GFLOAT or HFLOAT.

FLOPOIERR, floating point error or overflow

Explanation: WARNING – The program contains a numeric expression whose value is outside the valid range for floating-point numbers.

User Action: Modify the expression so that its value is within the allowable range.

FORFEEMUS, FORM FEED must appear at end of line

Explanation: ERROR – 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.

FORPARINC, formal parameter <name> inconsistent with actual

Explanation: An actual parameter in a DEF function invocation does not agree in data type with the formal parameter in the DEF statement.

User Action: Change the actual parameter in the function invocation to match the data type of the formal parameter in the DEF statement.

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 parameter 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 FIELD statement or the parameter list.

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.

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. 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 may 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.

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 encountered in a module that is not a FUNCTION subprogram.

User Action: Remove the EXIT FUNCTION or FUNCTIONEXIT statement.

FUNWITFUN, end of FUNCTION while not in FUNCTION

Explanation: ERROR – The program contains a FUNCTIONEND or END FUNCTION statement without an accompanying FUNCTION statement.

User Action: Supply a function subprogram or remove the FUNCTIONEND statement.

IDEMAYAPP, IDENT directive may appear only once per module

Explanation: ERROR – 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: FATAL – The quoted string in a %IDENT directive is too long.

User Action: Reduce the length of the string. In PDP–11 BASIC–PLUS–2, the maximum length is 6 characters. In VAX–11 BASIC, the maximum length is 31 characters.

IF_EXPMUS, IF directive expression must be terminated by %THEN

Explanation: FATAL – 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: FATAL – 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 EXPLICIT UNLOCK clause in the OPEN statement.

ILLARGBP2, illegal argument count for BP2

Explanation: INFORMATION – The program contains a SUB, DEF, or EXTERNAL FUNCTION reference with more than eight parameters. This error is reported only when the /FLAG:BP2COMPATIBILITY qualifier is in effect.

User Action: If the program must run under both VAX–11 BASIC and PDP–11 BASIC–PLUS–2, the function must have eight 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 proper parameter-passing mechanisms.

ILLCHA, illegal character <ASCII code>

Explanation: WARNING – The program contains illegal or incorrect characters.

User Action: Examine the program for correct usage of the 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 on PDP–11 systems can use only printable ASCII characters: ASCII values in the range 32 to 126, inclusive.

ILLCHAIDE, illegal character <ASCII value> in IDENT directive

Explanation: FATAL – 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: WARNING – 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.

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.

ILLIDEPDP, illegal %IDENT string for PDP–11

Explanation: ERROR – 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 between 1 and 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 through 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 between 1 and 32767, 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: WARNING – The program attempts matrix division. The operation is treated as a MAT multiply, and the compilation continues.

User Action: Remove the attempted matrix division. BASIC does not support this operation.

ILLMCHPDP, illegal passing mechanism on PDP–11s

Explanation: WARNING – 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 *BASIC Reference Manual* for allowable BASIC–PLUS–2 parameter-passing mechanisms.

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.

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.

ILLSTROPE, illegal string operator

Explanation: ERROR – The program has an invalid string operation, for example, A$ = B$ – C$.

User Action: Replace the invalid operator.

ILLUSAFIE, illegal usage of FIELDed variable

Explanation: ERROR – Either: 1) A MOVE TO or MOVE FROM statement contains a string variable or element of a string array that also appears in a FIELD statement, or 2) a MAT statement operates on an element of a string array that appears in a FIELD statement.

User Action: Either: 1) remove the variable from the FIELD statement or the MOVE statement or 2) remove the array from the MAT statement.

ILLUSEUNA, illegal use of unary operator

Explanation: FATAL – A compiler directive contains an invalid lexical expression, for example, %IF 1 – – 2, or %IF 1 NOT 2.

User Action: Correct the invalid lexical expression.

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.

IMPDECNOT implied declaration not allowed for <name> with /EXPLICIT

Explanation: ERROR – A program compiled with the /EXPLICIT qualifier contains an implicitly declared variable.

User Action: Declare the variable explicitly or compile the program without the /EXPLICIT qualifier.

INACODFOL, inaccessible code follows line <n> statement <m>

Explanation: WARNING – The program contains one or more statements that cannot be accessed, for example, a multi-statement line whose first statement is a GOTO, EXIT, ITERATE, RESUME, or RETURN.

User Action: Make sure that these statements are the only statements on the line, or the last statement on a multi-statement line.

INCDIRSYN, INCLUDE directive syntax error

Explanation: FATAL – A %INCLUDE directive either is not followed by a quoted string or incorrectly uses the %FROM %CDD clause.

User Action: Supply either a quoted string or the correct syntax for the %FROM %CDD clause.

INCFILMUS, INCLUDE directive file must be on a random access device

Explanation: ERROR – A %INCLUDE directive specifies a device other than a disk.

User Action: Change the %INCLUDE directive to specify a random access device.

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: FATAL – 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.

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. BASIC assumes a keyword is missing and continues compilation under that assumption so that other errors may 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. BASIC assumes a keyword is misspelled and continues compilation under that assumption so that other errors may 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, an HFLOAT 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 BASIC compiler.

User Action: Please submit an SPR with the source code of a small program that produces this error.

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.

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. The maximum in BASIC–PLUS–2 is 32767; the maximum in VAX–11 BASIC is 2147483467.

User Action: Supply a value in the valid range.

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.

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, or LONG values.

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, 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 a 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.

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; you cannot transfer control out of a DEF block except by an EXIT DEF, FNEXIT, FNEND, or END DEF statement.

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.

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 integer or string

Explanation: ERROR – A FIND or GET statement on an indexed file contains a key specification that is not an integer or string.

User Action: Change the key specification to be an integer or a string.

KEYMUSTBE, key, <vbl-name> in map <map-name> must be either integer or string

Explanation: ERROR – An OPEN statement contains a key specification that is not an unsubscripted integer or string variable.

User Action: Change the key specification to be an unsubscripted integer or string variable.

KEYNOTALL, key not allowed unless ORGANIZATION INDEXED

Explanation: ERROR – An OPEN statement for a nonindexed file contains a KEY clause.

User Action: Remove the KEY clause. You can specify keys only for indexed files.

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.

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.

LABNOTALL, label not allowed on RESUME

Explanation: ERROR – A RESUME statement specifies a label rather than a line number.

User Action: Change the label to a line number.

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 VAX–11 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.

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 VAX–11 BASIC and PDP–11 BASIC–PLUS–2, you must remove the incompatible language feature.

LANFEAOPE, language feature is operating system dependent

Explanation: ERROR – The program contains a PRINT statement with a RECORD clause on a system that does not support the RECORD clause.

User Action: Remove the RECORD clause.

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: FATAL – 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: INFORMATIONAL – A program compiled with the /ANSI_STANDARD qualifier contains an assignment statement that omits the LET keyword.

User Action: Supply a LET keyword.

LEXIDEMUS, lexical identifier must be declared before reference

Explanation: FATAL – A %IF directive names a lexical constant that was not named in a preceding %LET directive.

User Action: Declare the lexical constant with the %LET directive before referencing it.

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: FATAL – 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.

LOGOPENON, logical operation on non-integer 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 expression

Explanation: ERROR – A FOR statement attempts to assign a string value to a loop control variable.

User Action: Remove the string expression. You can assign only numeric values to the loop control variable.

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.

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: ERROR – The size of the MAP referenced in an OPEN statement is greater than 32767 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.

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.

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 on the program.

User Action: Reduce the number of nested %IF–%THEN–%ELSE–%END–%IF directives.

MAXDIMEXC, maximum number of dimensions exceeded for <array>. Maximum is <number>

Explanation: ERROR – An array declaration specifies more than the allowed number of dimensions.

User Action: Reduce the number of dimensions. The maximum is 8 on PDP–11 systems and 32 on VAX/VMS systems.

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.

MERGE, merged <item> and <item>

Explanation: ERROR – The program contains a syntax error. BASIC assumes that there is an incorrect space, for example, PR INT. Compilation continues so that other errors may 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.

MISENDIF, missing END IF directive

Explanation: FATAL – 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.

MULCHRARR, multiple character array name not ANSI

Explanation: INFORMATIONAL – 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: INFORMATIONAL – 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: FATAL – A lexical constant is named in more than one %LET directive.

User Action: Declare the lexical constant only once with %LET.

MULSTAPER, multiple statements per line not ANSI

Explanation: INFORMATIONAL – 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: ERROR – 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.

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.

NAMTOOLON, name is too long, changed to <name>

Explanation: WARNING – A variable or array name is longer than 31 characters. BASIC truncates the name to 31 characters and continues compilation so that other errors may 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.

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.

NODESCALL, no descriptor allocated for array <name>

Explanation: ERROR – An immediate mode statement required an array descriptor, but it was not available. BASIC allocates array descriptors only if the program code requires it.

User Action: None.

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.

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

Explanation: ERROR – The program attempted to use a feature that does not exist in this version of BASIC.

User Action: None.

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.

NUMARREXP, numeric array expected for CHANGE

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 CONSTANT A = "ABC"
```

User Action: Supply a numeric constant.

NUMIS_NEE, numeric expression is needed

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.

OPEEXPNOT, operator expected, not found

Explanation: 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.

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: ERROR – 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 VAX/VMS 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, WINDOWSIZE inconsistent with CLUSTERSIZE

Explanation: ERROR – An OPEN statement contains both a WINDOWSIZE and CLUSTERSIZE clause.

User Action: Remove either the WINDOWSIZE or CLUSTERSIZE clause. CLUSTERSIZE is valid only on RSTS/E systems, and WINDOWSIZE is valid on all systems except RSTS/E.

OPNREQIND, keyword requires INDEXED organization

Explanation: ERROR – An OPEN statement specifies a clause that is invalid for the file organization, for example, specifying CONNECT or a PRIMARY or ALTERNATE KEY for a relative or sequential file.

User Action: Remove the invalid clause.

OPNREQORG, keyword requires INDEXED or RELATIVE organization

Explanation: ERROR – An OPEN statement specifies a clause that is invalid for the file organization; for example, specifying BUCKETSIZE for a sequential file.

User Action: Remove the invalid clause.

OPNREQSEQ, keyword requires SEQUENTIAL organization

Explanation: ERROR – An OPEN statement specifies a clause that is invalid for the file organization, for example, specifying BLOCKSIZE, NOREWIND, or NOSPAN for a relative or indexed file.

User Action: Remove the invalid clause.

OPTCLACON, OPTION clause contradicts prior clause

Explanation: ERROR – The OPTION statement contains contradictory clauses, for example, specifying the default integer size as both BYTE and LONG.

User Action: Remove one of the clauses.

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.

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.

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 and/or DECIMAL data types.

OVRNOLINE, <keyword> overrides NOLINE

Explanation: WARNING – The program: 1) was compiled /NOLINES and 2) uses a keyword that requires line number information. For example, ERL and RESUME with line number both require that the program be compiled /LINES.

User Action: None. If you use a keyword that requires line number information, BASIC automatically overrides the /NOLINE qualifier.

PARCOUNOT, <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.

PARMODCHA, mode for parameter <n> of routine <name> changed to match declaration

Explanation: ERROR – The data type specified in a routine invocation does not match that of the routine declaration. 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.

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 /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 /EXPLICIT.

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; BASIC automatically passes parameters with the passing mechanism specified when the routine was declared.

PASMECNOT, passing mechanism not allowed for DEF

Explanation: ERROR – A DEF invocation specifies a passing mechanism for a parameter.

User Action: Remove the passing mechanism clause.

PASWITNO, <name> has a passing mechanism specified with no parameter list

Explanation: ERROR – 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 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.

PRICDDERR, prior severe CDD error

Explanation: ERROR – There have been one or more severe CDD errors, and this may be the reason for the following errors.

User Action: Recompile the program after correcting the first CDD–related error(s).

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.

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.

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 has FILL too large

Explanation: ERROR – The total size of a CDD record is greater than 65535 bytes.

User Action: Reduce the size of the record.

RECNOTDEF, RECORD <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.

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.

RECTOOBIG, record too big from INCLUDE directive file

Explanation: FATAL – The file specified in a %INCLUDE directive contains a record longer than 255 characters.

User Action: Edit the file to remove any records longer than 255 characters.

RECTOOLAR, RECORD <name> too large. Limit is 65535 bytes.

Explanation: ERROR – The components of a RECORD definition add up to more than 65535 bytes.

User Action: Reduce the size of the RECORD.

REGCLAONL, REGARDLESS clause only valid for GET

Explanation: ERROR – A REGARDLESS clause follows a FIND statement.

User Action: Remove the REGARDLESS clause from the FIND statement.

REPLACE, assuming <operator(s)> replaced by <operator>

Explanation: ERROR – The program contains a syntax error. BASIC found incorrect or multiple operators where another single operator makes more sense, for example, 10 A = = B. Compilation continues so that other errors may 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 numeric expression in a context requiring a string expression.

User Action: Supply a string 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.

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. BASIC cannot resequence lines in an INCLUDE file.

RESLINGTR, RESEQUENCE cannot generate line numbers greater than 32767

Explanation: ERROR – The RESEQUENCE command specified an interval or starting point that would have created a line number greater than 32767.

User Action: Reduce the interval or the starting point.

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.

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 non-numeric 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 /FLAG:OPTIMIZATION 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.

SCAOUTRAN, SCALE is out of range. Valid is 0 to 6.

Explanation: ERROR – The OPTION statement specifies a scale factor that is not between zero and six, inclusive.

User Action: Supply a valid scale factor.

SEVINTERR, severe internal error has been detected. Submit an SPR.

Explanation: ERROR – An error has been detected in the BASIC compiler.

User Action: Please submit an SPR with the source code of a small program that produces this error.

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. BASIC assumes that a keyword has been misspelled, and compilation continues so that other errors may 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: FATAL – 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.

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: INFORMATIONAL – A program compiled with the /ANSI_STANDARD qualifier contains a string array.

User Action: Remove the string array.

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: INFORMATIONAL – A program compiled with the /ANSI_STANDARD qualifier contains a string DEF.

User Action: Remove the string DEF.

STRIS_NEE, string expression is needed

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 DYNAMIC MAP variables

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. 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 non-numeric, 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. BASIC truncates the length specification to 512.

User Action: None. The maximum string length for virtual arrays is 512.

STROUTRAN, string is too large

Explanation: ERROR – A string exceeds the maximum allowable length. The maximum length in VAX–11 BASIC is 65535 characters. The maximum length in PDP–11 BASIC–PLUS–2 is 32767 characters.

User Action: Reduce the length of the string.

STRLITREQ, string literal required for compiler directive

Explanation: FATAL – 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.

STRVAREXP, string variable expected for CHANGE

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 references 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.

SUBWITSUB, end of SUB seen while not in SUB

Explanation: ERROR – A subprogram has a SUBEND or END SUB statement without a preceding SUB statement.

User Action: Supply a SUB statement as the first statement in the subprogram or delete the END SUB or SUBEND 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.

SUFNOTALL, suffix not allowed on variable <name>

Explanation: ERROR – A variable defined with explicit data type ends in a percent or dollar sign.

User Action: Remove the variable's percent or dollar sign.

TEXFOLEND, text following end of program unit must be on new BASIC line

Explanation: FATAL – The compiler detected text following an END, END SUB, or END FUNCTION statement.

User Action: Remove the text. In a multi-module source file, an END, END SUB, or END FUNCTION statement can be followed only by blank lines or numbered lines.

THEMUSFOL, THEN directive must follow a lexical expression

Explanation: FATAL – A %IF directive contains a lexical expression that is not immediately followed by a %THEN.

User Action: Supply a %THEN clause. %THEN, %ELSE, and %END %IF are required in a %IF directive.

TOOFEWARG, too few arguments

Explanation: ERROR – The invocation of a 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 a 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.

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.

UNDLINNUM, undefined line number

Explanation: ERROR – A statement tries to transfer control to a nonexistent line.

User Action: Replace the nonexistent line number with the correct destination line number.

UNLINCREA, UNLOCK EXPLICIT clause is 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.

UNSCDDLEV, unsupported CDD level <number>. Supported level is <number>.

Explanation: ERROR – The current CDD version is incompatible with BASIC.

User Action: Use a supported version of the CDD.

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.

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.

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: ERROR – 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.

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.

VIRARROVF, virtual array space exceeded at array <name>

Explanation: ERROR – The storage for virtual arrays on a single channel exceeds 2147483647 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.

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.

A.2 BASIC Environment Errors

ALLOCSML, allocated area may be too small for section

Explanation: ERROR – 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: 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.

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.

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. In the example, the complementary form of DOUBLE is SINGLE.

BADPROGNM, error in program name

Explanation: ERROR – The program name is longer than nine characters or contains non-alphanumeric characters.

User Action: Change the program name to be less than or equal to nine characters and make sure that it contains only letters and digits.

CANCON, can't continue

Explanation: ERROR – 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.

CHANGES, unsaved change has been made, CTRL–Z or EXIT to exit

Explanation: WARNING – A BASIC source program in memory has been modified, and an EXIT command or CTRL/Z has been typed. BASIC signals the error notifying you that if you exit from the compiler, the program modifications will be lost.

User Action: If you want to save the program, type SAVE. If you do not want to save the program, type EXIT or CTRL/Z.

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 sizes of the areas differ.

User Action: Make the size of the COMMON or MAP areas equal in size in all modules.

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.

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.

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.

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.

LEXDIRIMM, directive not valid in immediate mode

Explanation: ERROR – A compiler directive was typed in the BASIC environment.

User Action: None. Compiler directives are invalid in immediate mode.

LINNUMERR, illegal line number

Explanation: ERROR – A line number outside the valid range was typed.

User Action: Enter only line numbers in the range 1 to 32767, inclusive.

MISSINGLN, non-continued statement has no line number near lin-num

Explanation: ERROR – A new line in the source file: 1) does not follow a line ending with an ampersand, 2) does not begin with a line number, and 3) does not start with a space or tab (specifying an automatic continuation line).

User Action: Either: 1) start automatic continuation lines with a space or tab, 2) use an ampersand as the last character of the preceding line, or 3) start the line with a line number.

NOBASFRAM, no BASIC frame on stack

Explanation: ERROR – 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.

NOEDIT, no change made

Explanation: WARNING – The search string in an EDIT command was not located in the text.

User Action: Enter a valid search string.

NOFRAME, compiled procedure is currently not active

Explanation: WARNING – A STOP statement or CTRL/C was encountered, and neither the executing procedure nor any of its callers was the source compiled as a result of the RUN command.

User Action: None; you cannot examine or modify a variable in immediate mode unless the currently compiled program unit is active. 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.

NOLNROOM, out of memory for line numbers

Explanation: ERROR – The program contains more line-numbered statements than BASIC allows.

User Action: Change the program so that it uses multi-statement lines instead of having each statement on its own line or split the program into one or more program units in separate files.

NOTRANS, no main program

Explanation: WARNING – When a RUN command was typed, only subroutines or functions were available. BASIC requires a main program to receive the transfer of control.

User Action: Supply a main program.

NOTXTROOM, out of memory for statement text

Explanation: ERROR – The program contains more text than BASIC allows.

User Action: Split the program into one or more program units.

OBJFAIL, failure in loading object file

Explanation: ERROR – 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 VAX–11 Common Run-Time Library, you must link the program at DCL level. If the object file is in a user-supplied library, use the LIBRARY command to install the missing object module. You can load only BASIC object modules.

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.

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.

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 /DOUBLE. You cannot use scaling when in single precision.

SEQERR, attempt to sequence over existing statement

Explanation: WARNING – A SEQUENCE command specifies a starting line number that already exists in the 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.

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.

UNDEFINED, unresolved/undefined symbols

Explanation: ERROR – A program executed in the 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 BASIC environment.

UNEXPEOF, unexpected end of file

Explanation: ERROR – The compiler encountered an end-of-file immediately after an ampersand continuation character.

User Action: Remove the ampersand continuation character or continue the line.

UNKCOMINP, unknown command input

Explanation: An attempt was made to enter an invalid or unknown command.

User Action: Enter the BASIC command correctly.

A.3 Shared Error Messages

In some cases, the compiler can signal an error that it did not detect. For example, if you try to read a nonexistent file into memory, BASIC signals an error that was actually detected by the Record Management Services software. This is called a shared error message because it can be generated by more than one software facility.

This type of error appears as:

- A BASIC error message
- An additional message generated by the facility detecting the error
- Other optional messages from the system, explaining the error's cause

The following error messages can be generated by either: 1) compiler commands or 2) program errors.

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.

BADVALUE, <text> is an invalid keyword value

Explanation: FATAL – The command supplied an invalid value for a keyword.

User Action: Supply a valid value.

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.

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.

NOVALUE, <text> keyword requires a value

Explanation: ERROR – A keyword command was typed without a value.

User Action: Supply a valid keyword value.

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.

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.

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.

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.

A.4 Informational Messages from Flaggers

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.

Appendix B
Run-Time Error Messages

BASIC returns run-time error messages if an error occurs while a program is executing. BASIC diagnoses run-time errors and indicates the program line that generated the error. Warning error messages indicate that an error has occurred, but program execution continues. In some cases, BASIC reprompts for more information or correct data; in other cases, BASIC performs the specified operation, but the results are not as expected. Fatal error messages indicate that the program has aborted. You can recover from most fatal errors by including error-handling routines in your program. You do not need error-handling routines to trap errors that generate warning messages.

Section B.1 of this appendix lists VAX–11 BASIC run-time errors, alphabetized by mnemonic code. Section B.2 is a cross reference numerical listing of run-time errors generated by VAX–11 BASIC; Section B.3 lists error messages which VAX–11 BASIC does not generate but which can be displayed with the ERT$ function.

B.1 VAX–11 BASIC Run-Time Errors by Mnemonic

The VAX–11 BASIC error message format is:

```
%BAS–<l>–<mnemonic>, <message>
–BAS–I–FROLINMOD, from Line x in module y
```

where:

<l> Is a letter indicating the severity of the error. The severity indicator can be:

- 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, BASIC reprompts for more information or correct data; in other cases, BASIC performs the specified operation, but the results are not as expected. Fatal error messages indicate that the program has aborted. You can recover from most fatal errors by including error-handling routines in your program.

ARGDONMAT, Arguments don't match (ERR=88)

Explanation: ERROR – The arguments in a function call do not match the arguments defined for the function, either in number or in type.

User Action: Change the arguments in the function call to match those in the DEF, or change the arguments in the DEF.

ARGTOOLAR, Argument too large in EXP (ERR=49)

Explanation: ERROR – 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: ERROR – The program attempts to perform matrix addition or subtraction on input arrays with different dimensions.

User Action: Use arrays that have identical dimensions.

ARRMUSSQU, Arrays must be square (ERR=239)

Explanation: ERROR – The program attempts matrix inversion (MAT INV) on an array that is not square.

User Action: Use only square arrays when performing a matrix inversion.

BADDIRDEV, Bad directory for device (ERR=1)

Explanation: ERROR – The device directory does not exist or is unreadable.

User Action: Supply a valid directory.

BADRECIDE, Bad record identifier (ERR=143)

Explanation: ERROR – 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: ERROR – The value in the RECORDSIZE clause in the OPEN statement either: 1) is zero or greater than 16384 or 2) does not match the recordsize of an existing file.

User Action: Change the value in the RECORDSIZE clause.

CANCHAARR, Cannot change array dimensions (ERR = 240)

Explanation: ERROR – The program attempts to redimension an array to a different number of dimensions.

User Action: Change the array dimensions in the DIM or MAT statement.

CANFINFIL, Can't find file or account (ERR = 5)

Explanation: ERROR – The specified file or directory is not on the device.

User Action: Supply a valid file specification.

CANINVMAT, Can't invert matrix (ERR = 56)

Explanation: ERROR – 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: Check the file's protection. Include an access string for network file access.

CORFILSTR, Corrupted file structure (ERR = 29)

Explanation: ERROR – RMS has detected an invalid file structure on disk.

User Action: See your system manager.

DATFORERR, Data format error (ERR = 50)

Explanation: WARNING – The program specifies a data type in an INPUT or READ statement that does not agree with the value supplied.

User Action: Change the INPUT or READ statement or supply data of the correct type.

DATTYPERR, Data type error (ERR = 101)

Explanation: ERROR – The program attempts to access a parameter passed BY DESC (by descriptor), and the descriptor contains an incorrect data type.

User Action: Check the program code that created the passed parameter and make sure it creates a parameter of the correct data type.

DECERR, DECIMAL error or overflow (ERR = 181)

Explanation: ERROR – 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: ERROR – 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.

DIFUSELON, Differing use of LONG/WORD or SINGLE/DOUBLE qualifiers (ERR=229)

Explanation: ERROR – The main and subprograms were compiled with different LONG/WORD modes.

User Action: Recompile one of the programs with the same qualifier as the other.

DIRERR, Directive error (ERR=253)

Explanation: ERROR – A system service call resulted in an error.

User Action: See the *VAX/VMS I/O User's Guide* or the *VAX–11 Record Management Services Reference Manual.*

DIVBY_ZER, Division by 0 (ERR=61)

Explanation: ERROR – 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: ERROR – 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 re-create the file specifying DUPLICATES for that key.

ENDFILDEV, End of file on device (ERR=11)

Explanation: ERROR – 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.

ERRFILCOR, Error on OPEN – file corrupted (ERR=178)

Explanation: ERROR – The program attempted to open an invalid structure on disk.

User Action: See your system manager.

ERRTRANEE, ERROR trap needs RESUME (ERR=246)

Explanation: ERROR – An error handler attempts to execute an END, END SUB, END FUNCTION, SUBEND, FUNCTIONEND, or FNEND statement without first executing a RESUME statement.

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: ERROR – 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 *VAX/VMS System Messages and Recovery Procedures Manual* for RMS errors or retry the operation.

FIEOVEBUF, FIELD overflows buffer (ERR=63)

Explanation: ERROR – 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: ERROR – The operating system's file handler reported an error to RMS.

User Action: See the *VAX/VMS I/O User's Guide* or the *VAX–11 Record Management Services Reference Manual.*

FILATTNOT, File attributes not matched (ERR=160)

Explanation: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – The program executes an END DEF or FNEND statement before executing a function call.

User Action: Check program logic to make sure that END DEF or FNEND statements are executed only in multi-line DEFs or remove the END DEF or FNEND statement.

ILLALLCLA, Illegal ALLOW clause (ERR = 168)

Explanation: ERROR – The value specified for the ALLOW clause (sharing) is illegal for the type of file organization.

User Action: Change the ALLOW clause argument.

ILLARGLOG, Illegal argument in LOG (ERR = 53)

Explanation: ERROR – The program contains a negative or zero argument to the LOG or LOG10 function.

User Action: Supply an argument in the valid range.

ILLBYTCOU, Illegal byte count for I/O (ERR = 31)

Explanation: ERROR – 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.

ILLEXIDEF, Illegal exit from DEF* (ERR = 245)

Explanation: ERROR – A multi-line DEF* contains a branch to an END, END SUB, END DEF, SUBEND, or FUNCTIONEND statement.

User Action: Change the program logic so that the program executes the multi-line 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: ERROR – A FIELDed variable is referenced after a non–BASIC subprogram closed the file associated with that variable.

User Action: Check program logic; do not reference the variable after the file has been closed.

ILLFILNAM, Illegal file name (ERR = 2)

Explanation: ERROR – A file name: 1) is too long, 2) is 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: ERROR – 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.

ILLIO_CHA, Illegal I/O channel (ERR=46)

Explanation: ERROR – The program specified an I/O channel outside the legal range.

User Action: Specify I/O channels in the range 1 to 99, inclusive.

ILLKEYATT, Illegal key attributes (ERR=137)

Explanation: ERROR – 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.

ILLNUM, Illegal number (ERR=52)

Explanation: ERROR – 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: ERROR – 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.

ILLRECACC, Illogical record accessing (ERR=152)

Explanation: ERROR – 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: ERROR – A record contains an invalid byte count field.

User Action: Use the DCL DUMP command to check the file for possible bad data.

ILLRECLOC, illegal record locking (ERR=187)

Explanation: ERROR – The program contains an ALLOW or REGARDLESS 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 EXPLICIT UNLOCK clause in the OPEN statement.

ILLRESSUB, Illegal RESUME to subroutine (ERR=247)

Explanation: ERROR – While in an error handler activated by an ON ERROR GO BACK, the error handler attempts to RESUME without a line number.

User Action: None; you cannot RESUME to a line in any program module except the one containing the error handler.

ILLSWIUSA, Illegal switch usage (ERR=67)

Explanation: ERROR – The program attempts an illegal SYS call.

User Action: See the appropriate RSTS/E SYS call documentation.

ILLSYSUSA, Illegal SYS usage() (ERR=18)

Explanation: ERROR – The program attempted an illegal SYS call.

User Action: See the appropriate RSTS/E SYS call documentation.

ILLUSADEV, Illegal usage for device (ERR=133)

Explanation: ERROR – 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.

IMASQUROO, Imaginary square roots (ERR=54)

Explanation: ERROR – 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: ERROR – After an error has occurred, a program's error handler calls another program, and the called program executes an ON ERROR GO BACK statement before clearing the error with a RESUME statement.

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: WARNING – 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: ERROR – The program contains an integer whose absolute value is greater than 255 in BYTE mode, 32767 in WORD mode, or 2147483647 in LONG mode.

User Action: Use an integer in the valid range for specified data type.

INVFILOPT, Invalid file options (ERR=139)

Explanation: ERROR – 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: ERROR – The program attempts to perform a GET, FIND, or RESTORE on an INDEXED file using an invalid KEY, for example, an alternate KEY that has not been defined.

User Action: Use a valid KEY in the GET, FIND, or RESTORE statement.

INVRFAFIE, Invalid RFA field (ERR=173)

Explanation: ERROR – During a FIND or GET by RFA, an invalid record's file address was contained in the RAB.

User Action: Supply a correct RFA field.

IO_CHAALR, I/O channel already open (ERR=7)

Explanation: ERROR – The program attempted to OPEN channel zero (the controlling terminal).

User Action: Remove the OPEN statement; channel zero is always open.

IO_CHANOT, I/O channel not open (ERR=9)

Explanation: ERROR – 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: ERROR – 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: ERROR – The key specification exceeds the maximum record size.

User Action: Reduce the size of the key specification.

KEYNOTCHA, Key not changeable (ERR=130)

Explanation: ERROR – 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: ERROR – 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: ERROR – No input was received during the execution of a INPUT, LINPUT, or INPUT LINE statement that was preceded by a WAIT statement.

User Action: None; you must supply input within the specified time.

MATDIMERR, Matrix dimension error (ERR=124)

Explanation: ERROR – The program:

- Attempts to assign more than two dimensions to an array
- Attempts to reference an array with fewer or more subscripts than there are dimensions in the array
- Attempts to redimension an array that cannot be redimensioned

User Action: Change the number of array subscripts. Reference the array using the correct number of dimensions, or change the array so that it can be redimensioned.

MAXMEMEXC, Maximum memory exceeded (ERR=126)

Explanation: ERROR – 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.

User Action: Reduce the amount of string or I/O buffer space, or split the program into two or more modules.

MEMMANVIO, Memory management violation (ERR=35)

Explanation: ERROR – The program attempted to read or write to a memory location to which it was not allowed access.

User Action: If the program was compiled with /NOCHECK, it may be exceeding an array bound; recompile with /CHECK. Otherwise, check the program logic.

MISSPEFEA, Missing special feature (ERR=66)

Explanation: ERROR – 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: ERROR – 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.

NAMACCNOW, Name or account now exists (ERR=16)

Explanation: ERROR – The program attempted to RENAME a file and a file with that name already exists.

User Action: Use the KILL statement to erase the old file before using RENAME to name the new file or use a different name.

NEGFILSTR, Negative fill or string length (ERR=166)

Explanation: ERROR – 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 not allowed (ERR=176)

Explanation: INFORMATION – The program attempted a zero or negative TAB. This error is signalled 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: ERROR – 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: ERROR – A file specification's node name contains a syntax error.

User Action: Supply a valid node name.

NOTENDFIL, Not at end of file (ERR = 149)

Explanation: ERROR – 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.

NOTENODAT, Not enough data in record (ERR = 59)

Explanation: ERROR – 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: ERROR – The program attempted to use a feature that does not exist in this version of BASIC, for example, TIME(4%).

User Action: Do not use the feature.

NOTRANACC, Not a random access device (ERR = 64)

Explanation: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – No user storage space exists on the specified device.

User Action: Delete files that are no longer needed.

ONEOR_TWO, One or two dimensions only (ERR = 102)

Explanation: ERROR – The program contains a MAT statement that attempts to assign more than two dimensions to an array.

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: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – A CTRL/C was typed at the controlling terminal.

User Action: None; however, you can trap this error with an error handler.

PROLOSSOR, Program lost–Sorry (ERR=103)

Explanation: ERROR – A fatal system error caused your program to be lost.

User Action: This error should never occur. Submit a Software Performance Report.

PROVIO, Protection violation (ERR=10)

Explanation: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – A record previously located by its Record File Address (RFA) has been deleted.

User Action: None.

RECNOTFOU, Record not found (ERR=155)

Explanation: ERROR – A random access GET or FIND was attempted on a deleted or non-existent record.

User Action: None.

RECNUMEXC, RECORD number exceeds maximum (ERR=147)

Explanation: ERROR – 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 BASIC; it is a default, or it was specified by a non–BASIC program when the file was created.

RECOVEMAP, RECORDSIZE overflows MAP buffer (ERR=185)

Explanation: ERROR – The OPEN statement specifies a RECORDSIZE value larger than the size of the MAP specified in the MAP clause.

User Action: Increase the size of the MAP to match the RECORDSIZE value.

REDARR, Redimensioned array (ERR=105)

Explanation: ERROR – 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 of elements.

REMOVEBUF, REMAP overflows buffer (ERR = 183)

Explanation: ERROR – 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.

RESNO_ERR, RESUME and no error (ERR = 104)

Explanation: ERROR – The program executes a RESUME statement outside of the error handling routine.

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: ERROR – The program executes a RETURN statement before a GOSUB.

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: ERROR – 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: ERROR – A subprogram was compiled with a different SCALE factor than that of the calling program.

User Action: Recompile one of the programs with a scale factor that matches the other.

SIZRECINV, Size of record invalid (ERR = 156)

Explanation: ERROR – The program contains a COUNT 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 recordsize for fixed format records

User Action: Supply a valid COUNT value.

STO, Stop (ERR = 123)

Explanation: INFORMATION – The program executed a STOP statement.

User Action: Continue execution by typing CONTINUE or terminate execution by typing EXIT.

STRTOOLON, String too long (ERR=227)

Explanation: ERROR – The program attempts to create a string longer than 65535 bytes.

User Action: Reduce the length of the string.

SUBOUTRAN, Subscript out of range (ERR=55)

Explanation: ERROR – 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.

SYNERR, Syntax error (ERR=98)

Explanation: WARNING – The program contains a syntax error.

User Action: Correct the displayed statement to correspond to the syntax rules of BASIC language elements.

TAPBOTDET, Tape BOT detected (ERR=129)

Explanation: ERROR – 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 labelled (ERR=146)

Explanation: ERROR – 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: ERROR – 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: ERROR – 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: ERROR – A function invocation or CALL passed fewer arguments than were defined in the function or subprogram.

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: WARNING – An INPUT statement did not find enough data in one line to satisfy all the specified variables. This error is signalled 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: ERROR – A function invocation or CALL passed more arguments than were expected.

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: INFORMATION – The user has given too many items in response to the INPUT statement. This error is only signalled for ANSI INPUT.

User Action: Supply the correct number of items to the INPUT statement or change the INPUT statement.

UNEFILDAT, unexpired file date (ERR=179)

Explanation: ERROR – The program attempts to delete a file whose expiration date has not yet passed.

User Action: None.

VIRARRDIS, Virtual array not on disk (ERR=43)

Explanation: ERROR – The program attempted to reference a virtual array on a nondisk device.

User Action: Virtual arrays must be on disk; change the file specification in the OPEN statement for this array.

VIRARROPE, Virtual array not yet open (ERR=45)

Explanation: ERROR – 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: ERROR – The program attempted to access a VIRTUAL file and the buffer size was not a multiple of 512 bytes.

User Action: Change the I/O buffer to be a multiple of 512 bytes.

WHA, What? (ERR=109)

Explanation: ERROR – A command or immediate mode statement could not be processed.

User Action: Check for illegal verbs or improper formats.

B.2 VAX–11 BASIC Run-Time Errors by Number

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

16 NAMACCNOW, Name or account now exists

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

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

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, Program lost–Sorry

104 RESNO_ERR, RESUME and no error

105 REDARR, Redimensioned array

109 WHA, What?

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

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

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
183	REMOVEBUF, REMAP overflows buffer
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

B.3 Errors Not Generated by VAX–11 BASIC

The following errors cannot be generated in VAX–11 BASIC. However, they can be displayed with the ERT$ function and are included for completeness.

Number	Text
3	?Account or device in use
6	?Not a valid device
8	?Device not available
13	?User data error on device
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
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
99	?String is needed
100	?Number is needed
106	%Inconsistent subscript use
107	?ON statement needs GOTO
108	?End of statement not seen
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
184–226	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

Appendix C
ASCII Codes and Data Representation

C.1 ASCII Character Codes

Table C–1: ASCII Codes

Decimal Code	8–Bit Hex Code	Character	Remarks
0	00	NUL	Null (tape feed)
1	01	SOH	Start of heading (^A)
2	02	STX	Start of text (end of address, ^B)
3	03	ETX	End of text (^C)
4	04	EOT	End of transmission (shuts off TWX machine, ^D)
5	05	ENQ	Enquiry (WRU, ^E)
6	06	ACK	Acknowledge (RU, ^F)
7	07	BEL	Bell (^G)
8	08	BS	Backspace (^H)
9	09	HT	Horizontal tabulation (^I)
10	0A	LF	Line feed (^J)
11	0B	VT	Vertical tabulation (^K)
12	0C	FF	Form feed (page, ^L)
13	0D	CR	Carriage return (^M)
14	0E	SO	Shift out (^N)
15	0F	SI	Shift in (^O)
16	10	DLE	Data link escape (^P)
17	11	DC1	Device control 1 (^Q)
18	12	DC2	Device control 2 (^R)
19	13	DC3	Device control 3 (^)
20	14	DC4	Device control 4 (^T)
21	15	NAK	Negative acknowledge (ERR, ^U)
22	16	SYN	Synchronous idle (^V)
23	17	ETB	End-of-transmission block (^W)
24	18	CAN	Cancel (^X)
25	19	EM	End of medium (^Y)

(continued on next page)

Table C–1: ASCII Codes (Cont.)

Decimal Code	8–Bit Hex Code	Character	Remarks
26	1A	SUB	Substitute (^Z)
27	1B	ESC	Escape (prefix of escape sequence)
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation point
34	22	"	Double quotation mark
35	23	#	Number sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(	Left (open) parenthesis
41	29	)	Right (close) parenthesis
42	2A	*	Asterisk
43	2B	+	Plus sign
44	2C	,	Comma
45	2D	-	Minus sign, hyphen
46	2E	.	Period (decimal point)
47	2F	/	Slash (slant)
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than (left angle bracket)
61	3D	=	Equal sign
62	3E	>	Greater than (right angle bracket)
63	3F	?	Question mark
64	40	@	Commercial at
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L

Table C–1: ASCII Codes (Cont.)

Decimal Code	8–Bit Hex Code	Character	Remarks
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Left square bracket
92	5C	\	Backslash (reverse slant)
93	5D	]	Right square bracket
94	5E	^	Circumflex (caret)
95	5F	_	Underscore (underline)
96	60	`	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Left brace
124	7C	|	Vertical line
125	7D	}	Right brace
126	7E	~	Tilde
127	7F	DEL	Delete (rubout)

Notes

1. ASCII is a 7–bit character code with an optional parity bit (8) added for many devices. Programs normally use seven bits internally, with the eighth bit being zero; the extra bit is either stripped (on input) or added by a device driver (on output) so the program will operate with either parity or nonparity generating devices. The eighth bit is reserved for future standardization.
2. The International Reference Version (IRV) of ISO Standard 646 is identical to the IRV in CCITT Recommendation V.3 (International Alphabet No. 5). The character sets are the same as ASCII except that the ASCII dollar sign (hexadecimal 24) is the international currency sign, which looks like ¤ .
3. ISO Standard 646 and CCITT V.3 also specify the structure for national character sets, of which ASCII is the U.S. national set. Certain specific characters are reserved for national use. These are the values and symbols:

Hex Value	IRV	ASCII
23	#	#
24	¤	$ (General currency symbol vs. dollar sign)
40	@	@
5B	[	[
5C	\	\
5D	]	]
5E	^	^
60	`	`
7B	{	{
7C	|	|
7D	}	}
7E	‾	~ (Overline vs. tilde)

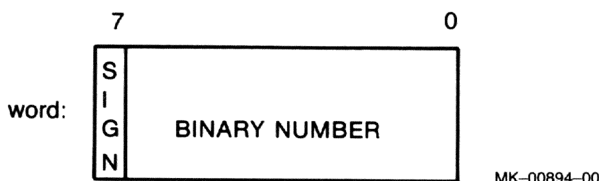
ISO Standard 646 and CCITT Recommendation V.3 (International Alphabet No. 5) are identical to ASCII except that: 1) the number sign (23) is represented as ## instead of #, and 2) certain characters are reserved for national use.

C.2 Integer Format

C.2.1 Byte-Length Integer Format

Byte-length integers are in the range –127 to 128 and are stored as a single byte starting on an arbitrary byte boundary. Bits are labeled from the right, zero through seven. See Figure C–1.

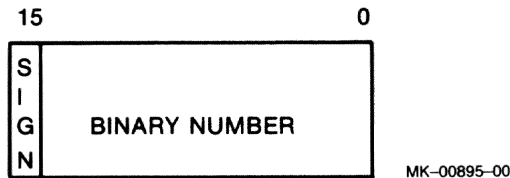
Figure C–1: Byte-Length Integer Format



C.2.2 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 through 15. See Figure C–2.

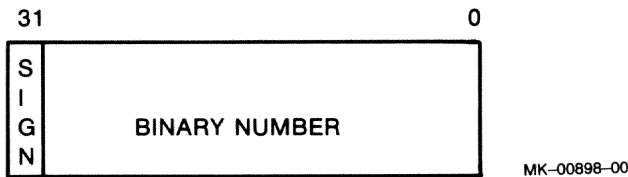
Figure C–2: Word-Length Integer Format



C.2.3 Longword Integer Format

Longword integers are stored as four contiguous bytes, starting on an arbitrary byte boundary. Values are in the range –2147483647 to 2147483647. See Figure C–3.

Figure C–3: Longword Integer Format



All integer types store data in two's complement format.

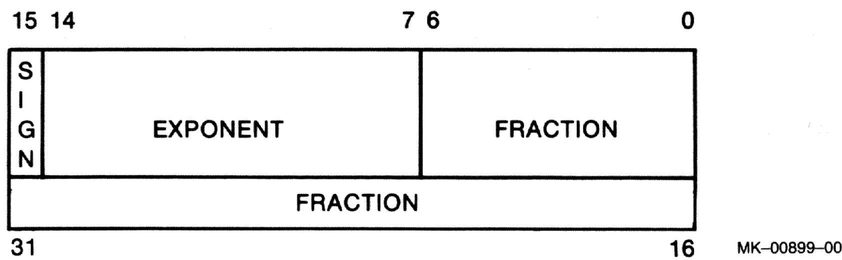
C.3 Real Number Formats

C.3.1 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 through 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 through 0 and 31 through 16 a normalized 24–bit fraction with the redundant most significant fraction bit not represented. See Figure C–4. The 8–bit exponent field encodes the values 0 through 255. 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 of 1 through 255 indicate true binary exponents of –127 through +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 $.29 * 10^{-38}$ through $* 10^{38}$. The precision of an F_floating number is approximately one part in 2^{23} (approximately 7 decimal digits).

Figure C–4: Single-Precision Real Number Format

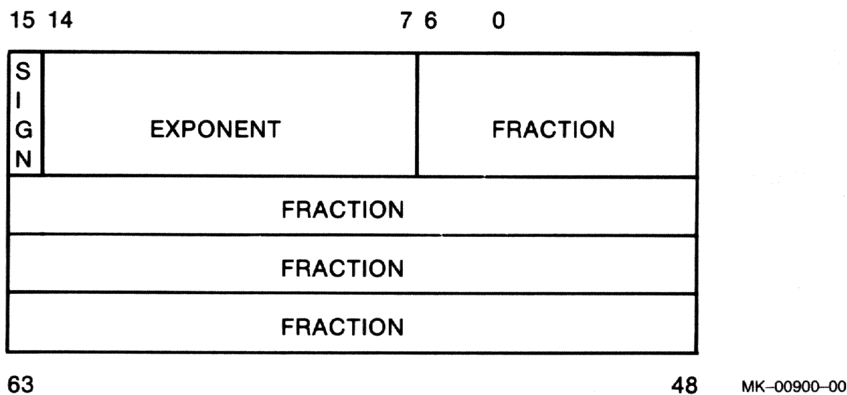


C.3.2 DOUBLE Floating-Point Number Format (D_floating)

Double-precision real number format is eight contiguous bytes starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 63. See Figure C–5. 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 through 47, 16 through 31, and 0 through 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 C–5: Double-Precision Real Number Format



The form of a double-precision real number is identical to that of a single-precision real number except for an additional 32 low-order bits. The exponent conventions and approximate range of values are the same as single-precision. The precision is approximately 16 decimal digits.

C.3.3 GFLOAT Floating-Point Number Format (G_floating)

The G_floating floating-point number format is eight contiguous bytes starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 63. The form of a G_floating number is sign magnitude with bit 15 the sign bit, bits 14 through 4 an excess–1024 binary exponent, and bits 3 through 0 and 63 through 16 a normalized 53–bit fraction with the redundant most significant fraction bit not represented. Within the fraction, the bits of increasing significance are 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11–bit exponent field encodes the values 0

through 2047. An exponent value of 0 together with a sign bit of 0 indicates that the G_floating number's value is 0. Exponent values of 1 through 2047 indicate true binary exponents of –1023 through +1023. 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 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).

C.3.4 HFLOAT Floating-Point Number Format (H_floating)

An H_floating floating-point number is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right 0 through 127. The form of an H_floating number is sign magnitude with bit 15 the sign bit, bits 14 to 0 an excess–16384 binary exponent, and bits 127 to 16 a normalized 113–bit fraction with the redundant most significant fraction bit not represented. Within the fraction, the bits of increasing significance are 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31. The 15–bit exponent field encodes the values 0 through 32767. An exponent value of 0 together with a sign bit of 0 indicates that the H_floating number has a value of 0. Exponent values of 1 through 32767 indicate true binary exponents of –16383 through +16383. 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 value of an H_floating number is in the approximate range $.84 * 10^{4932}$ through $.59 * 10^{4932}$. The precision of an H_floating number is approximately one part in 2^{112} (approximately 33 decimal digits).

C.4 Packed Decimal Number Format

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. Every 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 and sign is:

Digit or Sign	Decimal	Hex
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 + and 13 for –. The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range 0 through 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. Again, the length in bytes of the string is L/2 + 1. The value of a 0–length packed decimal string is identically 0; it contains only the sign byte that also includes the extra "0" digit.

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.

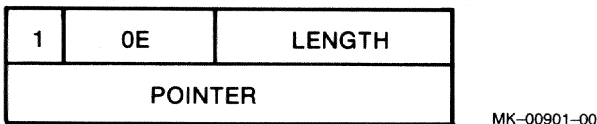
Note that the decimal point is specified by the descriptor for the packed decimal string. Refer to Section C.7.

C.5 String and Array Descriptor Format

C.5.1 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 a 14 (hexadecimal 0E; the VAX–11 code describing an ASCII character string). The fourth byte contains a 1. The second longword is a pointer containing the address of the string's first byte. See Figure C–6. See the *VAX–11 Procedure Calling and Condition Handling Standard* for more information.

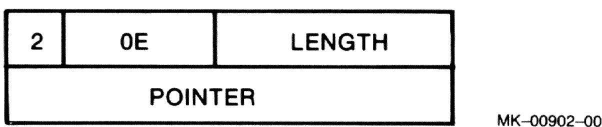
Figure C–6: Fixed-Length String Descriptor Format



C.5.2 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 a 14 (hexadecimal 0E; the VAX–11 code describing an ASCII character string). The fourth byte contains a 2. The second longword is a pointer containing the address of the string's first character. See Figure C–7.

Figure C–7: Dynamic String Descriptor Format



C.6 Array Descriptors

All array descriptors have the same format except for dynamic string arrays. A dynamic string array descriptor is actually an array of dynamic string descriptors.

Array descriptors have a 4–longword block of information, an optional 3–longword block of information, and an optional 4–longword block of information. For arrays created by BASIC, the optional blocks are always present.

In the first block, the first word of the first longword contains a value denoting the number of bytes in each array element (the value is 8 if it is a dynamic string array). The third byte contains a code describing the VAX–11 data type (the value is 24, hexadecimal 18, if it is a dynamic string array). The fourth byte is a four.

The second longword is a pointer containing the address of the first byte of data (or, if it is a dynamic string array, the address of the first element's descriptor).

The first byte of the third longword contains a scale factor (the negative of the number used in the SCALE command). The second byte contains a zero. The third byte contains array flags, including those specifying that blocks two and three are present (hexadecimal D0). The fourth byte contains a value equal to the number of dimensions in the array.

The fourth longword contains the total size of the array in bytes.

The first longword of the second block contains a value identical to the pointer (the second longword of the first block). The second longword contains a multiplier for the array's first dimension. The third longword is present only for two-dimensional arrays, and contains a multiplier for the array's second dimension.

The first longword of the third block contains a value equal to the lower bound of the array's first dimension. For BASIC arrays, this is always zero. The second longword contains a value equal to the upper bound of the array's first dimension. The third and fourth longwords are present only for two-dimensional arrays. The third longword contains a value equal to the lower bound of the array's second dimension. For BASIC arrays, this is always zero. The fourth longword contains the upper bound of the array's second dimension. See Figure C–8.

Figure C–8: Array Descriptor Format

4	DTYPE	LENGTH	
POINTER			
DIMCT	D0	00	SCALE
ARSIZE			

A0 (Same as POINTER)
M1
M2

L1
U1
L2
U2

MK–00903–00

A single descriptor form gives decimal size and scaling information for both scalar data and simple strings. See Figure D–10.

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 contains a signed power-of-10 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. The DIGITS field is 0; the number of digits is computed from the LENGTH field. The RESERVED field must be zero.

C.7 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 C–9.

Figure C–9: Decimal Scalar String Descriptor

9	21	LENGTH
POINTER		
RESERVED	DIGITS	SCALE

MK–00981–00

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 contains a signed power-of-10 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. The DIGITS field is 0; the number of digits is computed from the LENGTH field. The RESERVED field must be zero.

Appendix D
Example Programs

D.1 VAX–11 BASIC Record Sort

The following program consists of two include files and a BASIC source file:

SORT.INC:

```
!%TITLE "SORT.INC - Structures for interfacing with VAX-11 SORT/MERGE"
!%IDENT "X00.00"
!
!                         COPYRIGHT (c) 1982 BY
!             DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND  MAY  BE  USED  AND
! COPIED  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH
! THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR ANY
! OTHER  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAIL-
! ABLE TO ANY OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF THE  SOFT-
! WARE IS HEREBY TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT  NO-
! TICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIP-
! MENT CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY  OF
! ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!

!++
!
! FACILITY:
!
!   VAX-11 SORT documentation
!
! ABSTRACT:
!
!   Provides an example of an interface to  VAX-11  SORT/MERGE  from
!   VAX-11 BASIC.
!
```

(continued on next page)

```
! ENVIRONMENT:
!
!   VAX-11 user mode.
!
! AUTHOR: Brian Hetrick, CREATION DATE: 10 June 1982
!
! MODIFIED BY:
!
!       Brian Hetrick, 10-Jun-82: VERSION X00.00
! 000 - Original version of include file.
!
!--
%PAGE
!
! EQUATED SYMBOLS:
!

DECLARE WORD CONSTANT                     ! SORT key datatype constants   &
    SOR_K_DATATYPE_STRING = 1,            !  STRING                       &
    SOR_K_DATATYPE_INTEGER = 2,           !  INTEGER class                &
    SOR_K_DATATYPE_ZONED = 3,             !  Zoned character string       &
    SOR_K_DATATYPE_DECIMAL = 4,           !  DECIMAL class                &
    SOR_K_DATATYPE_UNSIGNED = 5,          !  Unsigned binary integer      &
    SOR_K_DATATYPE_LEADING_OVER = 6,!  Leading overpunch character  &
    SOR_K_DATATYPE_LEADING_SEP = 7, !  Leading separate character   &
    SOR_K_DATATYPE_TRAILING_OVER = 8,! Trailing overpunch character &
    SOR_K_DATATYPE_TRAILING_SEP = 9,!  Trailing overpunch character &
    SOR_K_DATATYPE_SINGLE = 10,           !  SINGLE                       &
    SOR_K_DATATYPE_DOUBLE = 11,           !  DOUBLE                       &
    SOR_K_DATATYPE_GFLOAT = 12,           !  GFLOAT                       &
    SOR_K_DATATYPE_HFLOAT = 13            !  HFLOAT
DECLARE WORD CONSTANT                     ! SORT key ordering constants   &
    SOR_K_ORDER_ASCENDING = 0,            !  Ascending key order          &
    SOR_K_ORDER_DESCENDING = 1            !  Descending key order
DECLARE BYTE CONSTANT                     ! SORT sort type constants      &
    SOR_K_SORTTYPE_RECORD = 1,            !  Record sort                  &
    SOR_K_SORTTYPE_TAG = 2,               !  Tag sort                     &
    SOR_K_SORTTYPE_INDEX = 3,             !  Index sort                   &
    SOR_K_SORTTYPE_ADDRESS = 4            !  Address sort

!
! LOCAL STORAGE:
!

!+
!   NOTE:  the user must declare the integer constant MAX_KEY_NUMBER
!   lexically before the %INCLUDE for this file, so that the  RECORD
!   Key_buffer_type can be defined.
!-

RECORD Key_descriptor_type                ! SORT key descriptor
    WORD Key_data_type_code               !  Data type code
    WORD Key_ordering_code                !  Ordering code
    WORD Key_start_position               !  Start position in record
    WORD Key_length                       !  Length
END RECORD Key_descriptor_type

RECORD Key_buffer_type                    ! SORT key buffer
    WORD Number_of_keys                   !  Number of keys
    Key_descriptor_type                                                   &
        Key_descriptor (Max_key_subscript)
END RECORD Key_buffer_type
```

PPREC.INC:

```
MAP (Opportunity_records)              ! Opportunity description file   &
    STRING                                                              &
        Opportunity_description = 64,! Description of opportunity       &
    WORD                                                                &
        Number_years,                  ! Number of years in cash flow   &
    SINGLE                                                              &
        Cash_flow (50)                 ! Cash flow per year
```

RECSORT.BAS:

```
10  %TITLE "RECSORT - Example of using VAX-11 SORT for record sort"
    %SBTTL "Overall description and modification history"
    %IDENT "X00.00"
    !
    !                       COPYRIGHT (c) 1982 BY
    !            DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
    !
    ! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND  MAY  BE  USED  AND
    ! COPIED  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH
    ! THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR ANY
    ! OTHER  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAIL-
    ! ABLE TO ANY OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF THE  SOFT-
    ! WARE IS HEREBY TRANSFERRED.
    !
    ! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT  NO-
    ! TICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIP-
    ! MENT CORPORATION.
    !
    ! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE  OR  RELIABILITY  OF
    ! ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
    !

    !++
    !
    ! FACILITY:
    !
    !   VAX-11 SORT documentation
    !
    ! ABSTRACT:
    !
    !   Provides an example of a record sort written in VAX-11 BASIC.
    !
    ! ENVIRONMENT:
    !
    !   VAX-11 user mode.
    !
    ! AUTHOR: Brian Hetrick, CREATION DATE: 10 June 1982
    !
    ! MODIFIED BY:
    !
    !       Brian Hetrick, 10-Jun-82: VERSION X00.00
    ! 000 - Original version of module.
    !
    !--
    %PAGE
    %SBTTL "Full description"
```

(continued on next page)

```
!++
!
! FUNCTIONAL DESCRIPTION:
!
!   Given a number of investment opportunities, computes the rate of
!   return on the opportunities, sorts the opportunities on the rate
!   of return, and reports on the opportunities in order of decreas-
!   ing rate of return.
!
!   The rate of return on an investment opportunity is  computed  as
!   the imputed interest rate giving a zero net present value on the
!   cash flow associated with the investment.
!
!   VAX-11 SORT is used to  sort  the  opportunities.   The  records
!   given  to  the  sort include only the computed interest rate and
!   the RFA of the input record describing the opportunity.
!
! FORMAL PARAMETERS:
!
!   None.
!
! IMPLICIT INPUTS:
!
!   The file pointed to by the logical name "INVEST_OPPORTUNITY".
!
!   This file must contain investment opportunity descriptions.  The
!   file must have records of the following format:
!
!   1.  Opportunity description.  This must be a 64-character  fixed
!       length ASCII character string.
!   2.  Number of years of opportunity cash flow.  This  must  be  a
!       WORD integer, and must be between 0 and 50 inclusive.
!   3.  One-dimensional array of cash flows.  This must be a  vector
!       of  SINGLE precision real values;  these give the cash flows
!       for year 0, year 1, ..., year n of the opportunity, where  n
!       is the number of years of the opportunity cash flow.
!
! IMPLICIT OUTPUTS:
!
!   The file pointed to by the logical name "INVEST_REPORT".
!
!   This file contains a report on the investment opportunities.
!
! COMPLETION CODES:
!
!   None.
!
! SIDE EFFECTS:
!
!   None.
!
!--
%PAGE
%SBTTL "Declarations"

!
! ENVIRONMENT SPECIFICATION:
!

OPTION                                                              &
    TYPE = EXPLICIT

!
! INCLUDE FILES:
!
```

```
!
! EQUATED SYMBOLS:
!

DECLARE LONG CONSTANT                                          &
    Max_Key_number = 1,            ! Maximum key number in sort    &
    Max_Key_subscript = 0,         ! Maximum key subscript for sort&
    True = -1,                     ! Logical TRUE for tests        &
    False = 0                      ! Logical FALSE for tests

%INCLUDE "SORT.INC"                ! Define symbols for SORT

!
! LOCAL STORAGE:
!

DECLARE                                                        &
                                                               &
    Key_buffer_type                                            &
        Key_description,           ! SORT key description block    &
                                                               &
    LONG                                                       &
        I,                         ! General counter               &
        Lines_per_page,            ! Lines per report page         &
        Opportunity_channel,       ! Channel for opportunity       &
                                   !  descriptions                 &
        Record_length,             ! Length of record from sort    &
        Report_channel,            ! Channel for report            &
        Return_code,               ! Return code from functions    &
                                                               &
    SINGLE                                                     &
        Current_guess_ROR,         ! Current ROR in iteration      &
        F,                         ! For Newton's method           &
        F_prime,                   ! For Newton's method           &
        Next_guess_ROR,            ! Next ROR in iteration         &
                                                               &
    STRING                                                     &
        Print_line                 ! Line for report

!
! GLOBAL STORAGE:
!

%INCLUDE "OPPREC.INC"              ! Opportunity record map area

MAP (Sort_record_release)          ! Communication with SORT       &
    SINGLE                                                     &
        Concatenated_keys,         !  Keys for sort                &
        Release_ROR,               !  Rate of return               &
    RFA                                                        &
        Release_RFA                !  RFA of description

MAP (Sort_record_release)          ! Communication with SORT       &
    STRING Sort_release_record = 14 !  String to pass to SORT

MAP (Sort_record_return)           ! Communication with SORT       &
    SINGLE                                                     &
        Opportunity_ROR,           !  Rate of return               &
    RFA                                                        &
        Opportunity_RFA            !  RFA of description

MAP (Sort_record_return)           ! Communication with SORT       &
    STRING Sort_return_record = 10 !  String to pass to SORT
```

(continued on next page)

```
!
! EXTERNAL REFERENCES:
!

EXTERNAL LONG FUNCTION                                                      &
                                                                            &
    LIB$FREE_LUN                        ! Release channel number            &
        (LONG BY REF),                  !  Channel to free                  &
                                                                            &
    LIB$GET_LUN                         ! Allocate channel number           &
        (LONG BY REF),                  !  Channel to allocate              &
                                                                            &
    LIB$LP_LINES                        ! Find number of lines per page     &
        (),                             !  No parameters                    &
                                                                            &
    SOR$INIT_SORT                       ! Initialize sort                   &
        (Key_buffer_type BY REF,        !  Key description block            &
         WORD BY REF,                   !  Longest record length            &
         LONG BY REF,                   !  Input file size in blocks        &
         BYTE BY REF,                   !  Number of work files             &
         BYTE BY REF,                   !  Type of sort                     &
         BYTE BY REF,                   !  Total key size                   &
         LONG BY VALUE,                 !  Comparision routine addr         &
         LONG BY REF),                  !  Sorting flags                    &
                                                                            &
    SOR$RELEASE_REC                     ! Release record to sort            &
        (STRING BY DESC),               !  Whole sort buffer                &
                                                                            &
    SOR$RETURN_REC                      ! Retrieve record from sort         &
        (STRING BY DESC,                !  Whole sort buffer                &
         LONG BY REF),                  !  Length of record                 &
                                                                            &
    SOR$SORT_MERGE                      ! Do the sort                       &
        ()                              !  No parameters

!
! INTERNAL REFERENCES:
!

DECLARE SINGLE FUNCTION                                                     &
                                                                            &
    Relative_error                      ! Approximation relative error      &
        (SINGLE,                        !  takes two singles                &
         SINGLE)                        !  and return relative error

DECLARE STRING FUNCTION                                                     &
                                                                            &
    Hex_integer                         ! Expresses an integer in hex       &
        (LONG)                          !  takes one long to express
    %PAGE
    %SBTTL "Environment initialization"
    !+
    !   Set up global error handler
    !-

    ON ERROR GO TO 31000

    !+
    !   Find number of lines per report page
    !-

    Lines_per_page = LIB$LP_LINES - '8'L
    %PAGE
    %SBTTL "Initialize"
```

```
Initialize_files:

    !+
    !   Get appropriate channel numbers.
    !-

    IF ('1'L AND LIB$GET_LUN (Opportunity_channel)) <> '1'L
    THEN
        GO TO Cannot_allocate_channels
    END IF

    IF ('1'L AND LIB$GET_LUN (Report_channel)) <> '1'L
    THEN
        CALL LIB$FREE_LUN (Opportunity_channel)
        GO TO Cannot_allocate_channels
    END IF

    !+
    !   Open the INVEST_OPPORTUNITY and INVEST_REPORT files.
    !-

    OPEN "INVEST_OPPORTUNITY" FOR INPUT                                     &
        AS FILE # Opportunity_channel,                                      &
        ORGANIZATION SEQUENTIAL,                                            &
        ALLOW READ,                                                         &
        ACCESS READ,                                                        &
        MAP Opportunity_records

    OPEN "INVEST_REPORT" FOR OUTPUT                                         &
        AS FILE # Report_channel,                                           &
        ORGANIZATION SEQUENTIAL VARIABLE,                                   &
        ALLOW NONE,                                                         &
        ACCESS WRITE,                                                       &
        RECORDSIZE 132

Initialize_sort:

    Key_description :: Number_of_keys = 1
    Key_description :: Key_descriptor (0) :: Key_data_type_code =          &
        SOR_K_DATATYPE_SINGLE
    Key_description :: Key_descriptor (0) :: Key_ordering_code =           &
        SOR_K_ORDER_DESCENDING
    Key_description :: Key_descriptor (0) :: Key_start_position =          &
        '1'W
    Key_description :: Key_descriptor (0) :: Key_length =                  &
        '4'W

    Return_code = SOR$INIT_SORT (                                           &
        Key_description,                ! Key buffer address                &
        '10'W,                          ! Longest record length             &
        ,                               ! Input file size                   &
        ,                               ! Number of work files              &
        ,                               ! Type of sort                      &
        '4'B,                           ! Total key size                    &
        ,                               ! Comparison routine address        &
        )                               ! Various flags

    IF (Return_code AND '1'L) <> '1'L
    THEN
        GO TO Cannot_initialize_sort
    END IF
    %PAGE
    %SBTTL "Get investment opportunities"
```

(continued on next page)

```
Get_opportunities:

     !+
     !   Read the opportunities one at a time until there are no more.
     !-

     WHILE True

          GET # Opportunity_channel
          Release_RFA = GETRFA (Opportunity_channel)

          !+
          !   Figure out the ROR.
          !
          !   The ROR is given by x when
          !
          !                           -1            -2                     -n
          !        0 = a + a (1+x)     + a (1+x)     + ... + a (1+x)
          !             0   1             2                   n
          !
          !   that is, when the net current value of the investment at
          !   interest rate x is equal to zero.  The right  hand  side
          !   of the equation is considered to be a function f(x), and
          !   its root is found by Newton's method;  this method iter-
          !   ates to find a zero through the equation
          !
          !   x     = x   - f(x )/f'(x )
          !    i+1     i       i      i
          !
          !   where f'(x) is the derivative of f with  respect  to  x.
          !   In this case, the derivative is
          !
          !                       -2            -3                  -n-1
          !   f'(x) = -a (1+x)     - 2a (1+x)     - ... - na (1+x)
          !             1              2                    n
          !
          !   The convergence test used is that the absolute value  of
          !   two successive estimates not exceed 0.0001 of the larger
          !   of the two successive estimates (0.01% relative error).
          !
          !   Horner's rule is used to  calculate  the  polynomial  in
          !   (1+x),  to  avoid  exponentiation.  Horner's rule states
          !   that the polynomial
          !
          !      n         n-1
          !   b y   + b     y    + ... + b y  + b
          !    n       n-1                1      0
          !
          !   can be evaluated as
          !
          !   ( ... ((b )y + b    )y + ... + b )y + b
          !            n      n-1             1      0
          !-

          Current_guess_ROR = 0
          Next_guess_ROR = 0.10

          WHILE Relative_error (Current_guess_ROR, Next_guess_ROR) > 0.0001

               Current_guess_ROR = Next_guess_ROR

               F = 0.0
               FOR I = Number_years TO 0 STEP -1
                    F = Cash_flow (I) + F / (1.0 + Current_guess_ROR)
                    NEXT I
```

```
            F_prime = 0.0
            FOR I = Number_years TO 1 STEP -1
                F_prime = F_prime / (1.0 + Current_guess_ROR) +           &
                    I * Cash_flow (I)
                NEXT I
            F_prime = - F_prime / (1.0 + Current_guess_ROR) ^ '2'L

            Next_guess_ROR = Current_guess_ROR - F / F_prime

            NEXT

        !+
        !   The opportunity ROR has been computed.
        !-

        Release_ROR = Next_guess_ROR
        Concatenated_Keys = Release_ROR

        !+
        !   Release the record to the sort.
        !-

        Return_code = SOR$RELEASE_REC (                                    &
            Sort_release_record)     ! The record to be sorted

        IF (Return_code AND '1'L) <> '1'L
        THEN
            GO TO Cannot_release_record
            END IF

        NEXT
    %PAGE
    %SBTTL "Do the sort"

Do_sort:

    !+
    !   All record have been read.  Do the sort.
    !-

    Return_code = SOR$SORT_MERGE
    IF (Return_code AND '1'L) <> '1'L
    THEN
        GO TO Cannot_do_sort
    END IF
    %PAGE
    %SBTTL "Retrieve records and report"

Retrieve_and_report:

    Return_code = SOR$RETURN_REC       ! Retrieve the first record      &
       (Sort_return_record,            !  Record returned by sort       &
        Record_length)                 !  Length of record

    WHILE (Return_code AND '1'L) = '1'L

        !+
        !   Retrieve the opportunity description
        !-

        GET # Opportunity_channel, RFA Opportunity_RFA

        !+
        !   Describe the opportunity
        !-
```

(continued on next page)

```
        Print_line = "Opportunity: " + Opportunity_description
        GOSUB Print_print_line
        Print_line = "    Rate of return: " +                          &
            FORMAT$ (100.0 * Opportunity_ROR, "###.##") + "%"
        GOSUB Print_print_line
        Print_line = "    Cash flow:        Year    Amount"
        GOSUB Print_print_line
        FOR I = 0 TO Number_years
            Print_line = HT + HT + FORMAT$ (I, "####") + HT +          &
                FORMAT$ (Cash_flow (I), "###,###.##")
            GOSUB Print_print_line
            NEXT I
        Print_line = STRING$ (25, ASCII ("-"))
        GOSUB Print_print_line

        !+
        !   Get the next record
        !-

        Return_code = SOR$RETURN_REC! Retrieve the first record       &
            (Sort_return_record,     !  Record returned by sort       &
             Record_length)          !  Length of record

        NEXT
    %PAGE
    %SBTTL "Shutdown"

Shut_down:

    !+
    !   Shut down the sort
    !-

    CALL SOR$END_SORT

    !+
    !   Close files.
    !-

Close_shut_down:

    CLOSE # Report_channel, Opportunity_channel

    !+
    !   Free the channels
    !-

Free_shut_down:

    CALL LIB$FREE_LUN (Report_channel)
    CALL LIB$FREE_LUN (Opportunity_channel)

    !+
    !   End the program
    !-

    GO TO 32767
    %PAGE
    %SBTTL "Internal subroutine - Print_print_line"
```

```
Print_print_line:

        !+
        !
        ! FUNCTIONAL DESCRIPTION:
        !
        !   Writes a print line to the file open on Report_channel.  May
        !   break pages and write page headings.
        !
        ! IMPLICIT INPUTS:
        !
        !   Lines_per_page - The number of lines per report page.
        !   Print_line - The text line to be printed.
        !   Print.line - The current line number on the page.
        !   Print.page - The current page number.
        !
        ! IMPLICIT OUTPUTS:
        !
        !   Print.line - The current line number on the page.
        !   Print.page - The current page number.
        !
        ! SIDE EFFECTS:
        !
        !   Writes records to the channel Report_channel
        !
        !-

        DECLARE                                                          &
            LONG                                                         &
                Print.line,                  ! Current line number       &
                Print.page                   ! Current page number

        !+
        !   Handle page break
        !-

        IF (Print.page = 0) OR (Print.line > Lines_per_page)
        THEN
            Print.page = Print.page + 1%
            PRINT # Report_channel,                                      &
                FF;                                                      &
                "Analysis of investment opportunities";                  &
                STRING$ (2%, ASCII (HT));                                &
                "Page";                                                  &
                FORMAT$ (Print.page, "###")
            PRINT # Report_channel
            Print.line = '2'L
        END IF

        !+
        !   Print the line
        !-

        PRINT # Report_channel, Print_line
        Print.line = Print.line + '1'L

        RETURN
        %PAGE
        %SBTTL "Internal function - Relative_error"

        DEF SINGLE Relative_error                                        &
            (SINGLE Real_number_1,           ! First number              &
             SINGLE Real_number_2)           ! Second number
```

(continued on next page)

```
!+
!
! FUNCTIONAL DESCRIPTION:
!
!    Computes the relative error between two numbers.
!
!    This is equivalent to:
!
!    abs (a-b) / max {abs (a), abs (b)}
!
!    Note that the operation is symmetric in its arguments.
!
! FORMAL PARAMETERS:
!
!    First_real - The first real number of the pair for which the
!         relative error is to be computed.
!    Second_real - The second real number of the pair  for  which
!         the relative error is to be computed.
!
! IMPLICIT INPUTS:
!
!    None.
!
! IMPLICIT OUTPUTS:
!
!    Numer.ator - Used as a local variable
!    Denom.inator - Used as a local variable
!
! FUNCTION VALUE:
!
!    The relative error of the two numbers.
!
! SIDE EFFECTS:
!
!    None.
!-

DECLARE                                                          &
    REAL                                                         &
        Numer.ator,        ! Numerator of relative error fraction  &
        Denom.inator       ! Denominator of relative error fraction

Numer.ator = ABS (Real_number_1 - Real_number_2)

Denom.inator = ABS (Real_number_1)
IF ABS (Real_number_2) > Denom.inator
THEN
    Denom.inator = ABS (Real_number_2)
END IF

IF Denom.inator = 0.0
THEN
    Relative_error = 0.0
ELSE
    Relative_error = Numer.ator / Denom.inator
END IF

END DEF
%PAGE
%SBTTL "Internal function - Hex_integer"

DEF STRING Hex_integer                                           &
   (LONG Integer_value)           ! Value to express in hex
```

```
!+
!
! FUNCTIONAL DESCRIPTION:
!
!   Finds the hexadecimal representation of a longword integer.
!
!   Uses OTS$CVT_L_TZ for the actual grunt work.
!
! FORMAL PARAMETERS:
!
!   Long_value - The longword to be expressed in hexadecimal.
!
! IMPLICIT INPUTS:
!
!   None.
!
! IMPLICIT OUTPUTS:
!
!   Hex.value - Used as a local variable
!
! FUNCTION VALUE:
!
!   A string expressing the argument in hexadecimal.
!
! SIDE EFFECTS:
!
!   None.
!-

DECLARE                                                               &
    STRING                                                            &
        Hex.value

Hex.value = STRING$ (8%, 0%)
CALL OTS$CVT_L_TZ (Integer_value, Hex.value, 8% BY VALUE)
Hex_integer = Hex.value
END DEF
%PAGE
%SBTTL "Error reporting"

Cannot_allocate_channels:

    PRINT "Cannot allocate channels for input, output files"
    GO TO 32767

Cannot_initialize_sort:

    PRINT "Cannot initialize sort, return code = ";                   &
        Hex_integer (Return_code)
    GO TO Close_shut_down

Cannot_release_record:

    PRINT "Cannot release record to sort, return code = ";            &
        Hex_integer (Return_code)
    GO TO Close_shut_down

Cannot_do_sort:

    PRINT "Cannot perform sort, return code = ";                      &
        Hex_integer (Return_code)
    GO TO Shut_down
    %PAGE
    %SBTTL "Common error handling"
```

(continued on next page)

```
31000   !+
        !   Common error handling:
        !-

        RESUME 31010

31010   !+
        !   Test for end of file on input
        !-

        IF ERR = '11'L
        THEN
             GO TO Do_sort
        END IF

        PRINT "Unknown error:"; ERT$ (ERR)
        PRINT "Aborting"

        GO TO 32767
        %PAGE
        %SBTTL "Module end"

32767   END
```

D.2 VAX–11 BASIC USEROPEN Routine

The following program consists of one %INCLUDE file and a FUNCTION subprogram:

FABRABDEF.BAS %INCLUDE File:

```
%NOLIST

!+
!        FABRABDEF.BAS
!
!        RMS Data Structures Definitions
!-

%LIST

%PAGE
%SBTTL   "File Access Block Definition"

         RECORD fab

              BYTE fab$b_bid
              BYTE fab$b_bln
              WORD fab$w_ifi
              LONG fab$l_fop
              LONG fab$l_sts
              LONG fab$l_stv
              LONG fab$l_alq
              WORD fab$w_deq
              BYTE fab$b_fac
              BYTE fab$b_shr
              LONG fab$l_ctx
              BYTE fab$b_org
              BYTE fab$b_rat
              BYTE fab$b_rfm
              LONG fab$l_jnl
              LONG fab$l_xab
              LONG fab$l_nam
              LONG fab$l_fna
              LONG fab$l_dna
```

```
            BYTE fab$b_fns
            BYTE fab$b_dns
            WORD fab$w_mrs
            LONG fab$l_mrn
            WORD fab$w_bls
            BYTE fab$b_bks
            BYTE fab$b_fsz
            LONG fab$l_dev
            LONG fab$l_sdc
            LONG fab$w_gbc
            STRING fill         = 6%

        END RECORD

    %PAGE
    %SBTTL  "Record Access Block Definition"

        RECORD rab

            BYTE rab$b_bid
            BYTE rab$b_bln
            WORD rab$b_isi
            LONG rab$l_rop
            LONG rab$l_sts
            LONG rab$l_stv
            RFA  rab$w_rfa
            LONG rab$l_ctx
            BYTE rab$b_rac
            BYTE rab$b_tmo
            WORD rab$w_usz
            WORD rab$w_rsz
            LONG rab$l_ubf
            LONG rab$l_rbf
            LONG rab$l_rhb
            VARIANT
            CASE
                LONG rab$l_kbf
                BYTE rab$b_ksz
            CASE
                LONG rab$l_pbf
                BYTE rab$b_psz
            END VARIANT
            BYTE rab$b_krf
            BYTE rab$b_mbf
            BYTE rab$b_mbc
            VARIANT
            CASE
                LONG rab$l_bkt
            CASE
                LONG rab$l_dct
            END VARIANT
            LONG rab$l_fab
            STRING fill         = 4%

        END RECORD
```

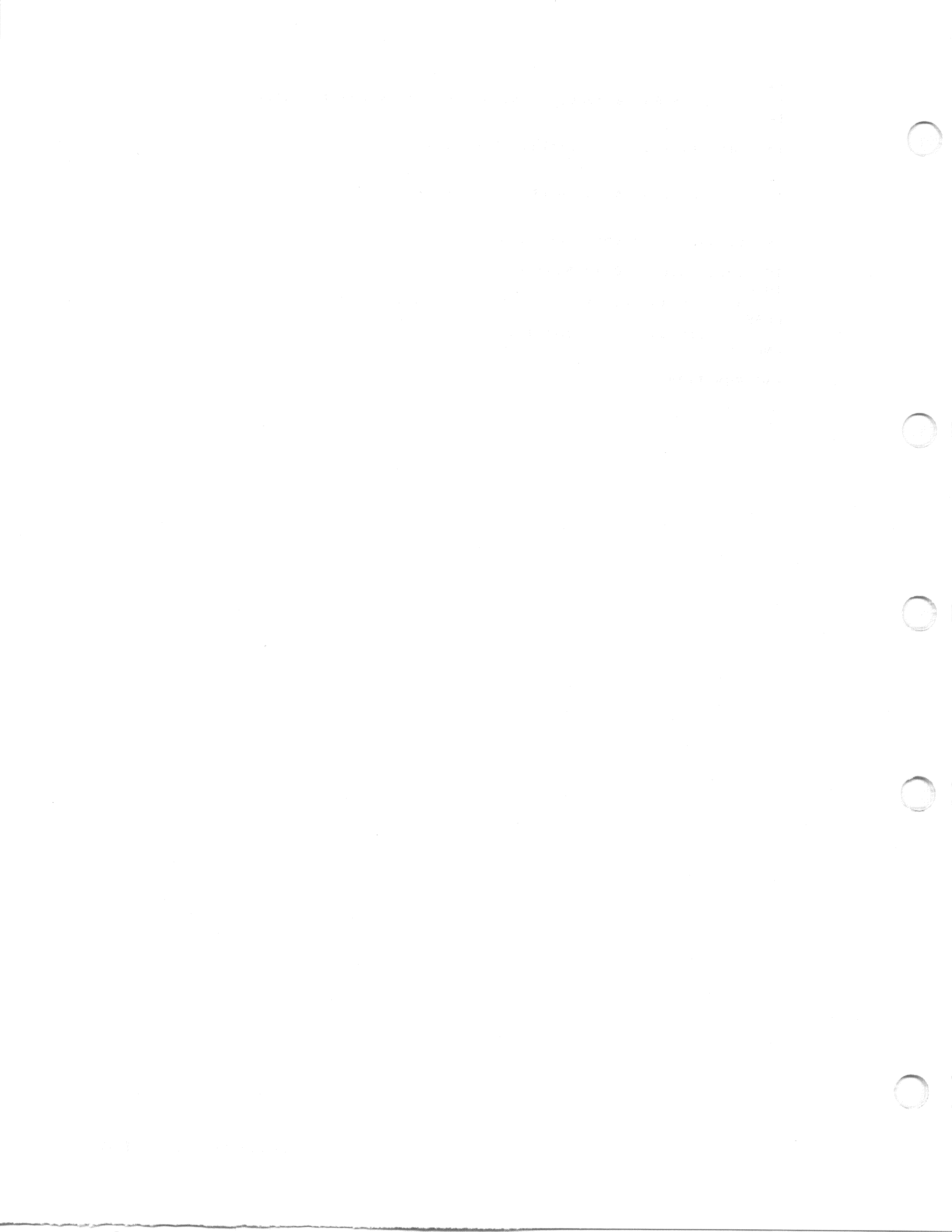
USEROPEN Program

```
1   %TITLE  "Get_RAB_Address"
    %SBTTL  "Useropen Routine to obtain RAB address"
    %IDENT  "Version 1.0"

    FUNCTION LONG get_rab_address (                                  &
                        fab user_fab,                                &
                        rab user_rab,                                &
                        long channel )
```

(continued on next page)

```
!++
!
! FUNCTIONAL DESCRIPTION:
!
!       Save the address of the RMS Record Access Block
!       allocated by the caller in a global symbol.
!       Call the RMS routines necessary to open the file
!       specified by the caller in the File Access Block.
!
! FORMAL PARAMETERS:
!
!       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.
!
! PRIVILEGES REQUIRED:      None
!
! EXTERNAL ROUTINES CALLED:
!
!       sys$open            RMS file open routine
!       sys$connect         RMS buffer connect routine
!
! MODIFICATION HISTORY:
!
!--

OPTION  INACTIVE = SETUP, TYPE = EXPLICIT

%PAGE
%SBTTL  "Routine and Storage Declarations"

!+
!       RMS Routines and RMS success code
!-

EXTERNAL LONG FUNCTION  sys$open( fab ),                  &
                        sys$connect( rab )

EXTERNAL LONG CONSTANT  rms$_normal

!+
!       Common area used to pass RAB address to caller.
!-

COMMON  (RAB_ptr)       LONG    rab_address

!+
!       Local storage
!-

DECLARE LONG            rms_status

%INCLUDE 'fabrabdef.bas'

%PAGE
%SBTTL  "Main Routine"
```

```
!+
!       Save RAB address in global symbol known to caller.
!-

rab_address = LOC(user_rab::rab$b_bid)

!+
!       Perform standard RMS open sequence
!-

rms_status = sys$open( user_fab )

IF  rms_status AND rms$_normal
THEN
    get_rab_address = sys$connect( user_rab )
ELSE
    get_rab_address = rms_status
END IF

END FUNCTION
```



Appendix E
Listing File Format

This appendix describes the compilation listing produced by VAX–11 BASIC.

```
(1) LISTER$MAIN                      Listing Tester                 13-AUG-1982 10:31:59   VAX-11 BASIC V2.0-0        Page   1
    V1.5                             Test                           13-AUG-1982 10:25:05   BAS$$DISK:[PARODI.BASIC]LISTER.BAS

   (2)   1 10       !This program only shows the format of a listing
         1          !file.  It does no useful work.
         1          %TITLE "Listing Tester"
         1          %SBTTL "Test"
         1          %IDENT "V1.5"
(3)      1          %INCLUDE "MAPS.DEF"
    I1   1          !MAP definition file
    I1   1          MAP (SHARED)    STRING A = 16,  &
    I1   1                          LONG B,         &
    I1   1                          DOUBLE C,       &
    I1   1                          BYTE D
    I1   2
    I1   2
         2          DECLARE INTEGER INDEX
         3          DECLARE LONG CONSTANT TRUE = -1
         4          DECLARE SINGLE Q(5)
         5
         5          %IF %VARIANT = 2
(4)      5          %THEN
      F1 5                  DECLARE DOUBLE Z(10)
         5          %END %IF
         5
         5
         5     First_loop:
         5          FOR INDEX = 0 TO 5
         6            PRINT Q(INDEX)
         7          NEXT INDEX
         8
         8     Second_loop:
         8          WHILE TRUE
         9            INDEX = INDEX + 1
        10            EXIT Second_loop IF INDEX => 5
        11          NEXT
        11
         1 32767    END
         1
```

```
LISTER$MAIN                    Listing Tester                  13-AUG-1982 10:31:59  VAX-11 BASIC V2.0-0        Page   2
Cross Reference                                                13-AUG-1982 10:25:05  BAS$$DISK:[PARODI.BASIC]LISTER.BAS
```

(5)

```
User Identifier Cross Reference

Symbol                                   Datatype   Name Type

        +---------------------------------------------------+
        !    #  Defining reference                          !
        !    @  Destructive reference                       !
        !    P  Parameter reference                         !
        !    R  Redefining reference                        !
        +---------------------------------------------------+

A                                        STR=16     MAP            SHARED + 0
            10.1#

B                                        LONG       MAP            SHARED + 16
            10.1#

C                                        DOUBLE     MAP            SHARED + 20
            10.1#

D                                        BYTE       MAP            SHARED + 28
            10.1#

INDEX                                    LONG
            10.2#          10.5            10.6            10.7            10.9@           10.10

Q()                                      SINGLE
            10.4#          10.6

TRUE                                     LONG       CONSTANT
            10.3#          10.8
```

```
LISTER$MAIN                    Listing Tester                  13-AUG-1982 10:31:59  VAX-11 BASIC V2.0-0        Page   3
Cross Reference                Map                             13-AUG-1982 10:25:05  BAS$$DISK:[PARODI.BASIC]LISTER.BAS
```

(6)

```
Map Cross Reference

Symbol                                   References

SHARED                                              MAP
            10.1#

A                                        STR=16     MAP            SHARED + 0
            10.1#

B                                        LONG       MAP            SHARED + 16
            10.1#

C                                        DOUBLE     MAP            SHARED + 20
            10.1#

D                                        BYTE       MAP            SHARED + 28
            10.1#
```

(7)

```
Label Cross Reference

Symbol                                   References

FIRST_LOOP
            10.5#

SECOND_LOOP
            10.8#          10.10
```

```
LISTER$MAIN                    Listing Tester                  13-AUG-1982 10:31:59  V2.0-0        Page   4
                               ALLOCATION MAP                  13-AUG-1982 10:25:05  BAS$$DISK:[PARODI.BASIC]LISTER.BAS
```

(8)

```
Allocation information for MAP SHARED

Name                                 Offset     Size      Type

A                                    0          16        Static string
B                                    16         4         Long
C                                    20         8         Double
D                                    28         1         Byte
```

(9)

```
Named constants

Name                                 Type           Value

TRUE                                 Long           -1
```

(10)
```
Allocation information for main program LISTER  Offset based on (R11)

Name                                    Offset     Size      Type

Variables for this program unit

INDEX                                   119        4         Long
Q                                       95         24        Single
    Dimensions : ( 5 )
```

(11)
```
PROGRAM SECTIONS

    Name                                   Bytes  Attributes

  0 $PDATA                                    92  PIC CON REL LCL   SHR NOEXE   RD NOWRT LONG
  1 $CODE                                    166  PIC CON REL LCL   SHR   EXE   RD NOWRT LONG
  2 $ARRAY                                     0  PIC CON REL LCL   SHR NOEXE   RD   WRT LONG
  3 $DESC                                      0  PIC CON REL LCL   SHR NOEXE   RD   WRT LONG
  4 SHARED                                    29  PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG
```

(12)
```
EXTERNAL REFERENCES

OTS$LINKAGE                   BAS$LINKAGE                   BAS$INIT_R8                  BAS$END_R8
BAS$PRINT                     BAS$IO_END                    BAS$OUT_F_V_B
```

```
LISTER$MAIN                   Listing Tester                  13-AUG-1982 10:31:59  VAX-11 BASIC V2.0-0         Page    5
Qualifier summary                                             13-AUG-1982 10:25:05  BAS$$DISK:[PARODI.BASIC]LISTER.BAS
```

(13)
```
DEFAULT DATA TYPE INFORMATION:                      LISTING FILE INFORMATION INCLUDES:
     Data type : REAL                                       Source
     Real size : SINGLE                                     Cross reference
     Integer size : LONG                                    CDD Definitions
     Decimal size : (15,2)                                  Environment
     Scale factor : 0                                    NO Override of %NOLIST
     NO Round decimal numbers                            NO Machine code
                                                            Map
COMPILATION QUALIFIERS IN EFFECT:                           INCLUDE files
     NO Object file
        Overflow check integers                     FLAGGERS:
        Overflow check decimal numbers                      Declining features
        Bounds checking                                  NO BASIC PLUS 2 subset
     NO Syntax checking
        Lines
        Variant : 0                                 DEBUG INFORMATION:
        Warnings                                            Traceback records
        Informationals                                   NO Debug symbol records
        Setup
        Object Libraries : BAS$$DISK:[PARODI.BASIC]TEST.OLB;
```

(14)
```
LISTER$MAIN                   Listing Tester                  3-AUG-1982 14:16:32  VAX-11 BASIC V2.0-0         Page    6
Generated code                                                3-AUG-1982 10:25:05  BAS$$DISK:[PARODI.BASIC]LISTER.BAS

                                    0000:         .TITLE  LISTER$MAIN
                                    0000:         .IDENT  V1.5
                                    0000:         .PSECT  $PDATA
                          00000048  0000:         .LONG   72
                          1C000102  0004:         .LONG   469762306
                          00000048  0008:         .LONG   72
                          00000000  000C:         .LONG   0
                          00000001  0010:         .LONG   1
                          00000000  0014:         .LONG   0
                          00000000  0018:         .LONG   0
                          00000000  001C:         .LONG   0
                          00000000  0020:         .LONG   0
                          00000000  0024:         .LONG   0
                          00000000  0028:         .LONG   0
                          00000018  002C:         .LONG   24
                          00000000  0030:         .LONG   0
                          00000000  0034:         .LONG   0
                          0000005B  0038:         .LONG   91
                          0000005B  003C:         .LONG   91
                          00000000  0040:         .LONG   0
                          0000005B  0044:         .LONG   91
                44 45 52 41 48 53 06  0048:       .ASCIC  "LISTER"
                          00000002  004F:         .LONG   2
                         0016 000A  0053:         .WORD   10,22
                         009A 7FFF  0057:         .WORD   32767,154
                                    005B:         ;       Decimal constants
                                0C  005B:         .PACKED +0
                                    005C:         ;       String constants
                                    005C:
                                    0000:         .PSECT  $CODE
                                    0000:  LISTER$MAIN::
                              CFFC  0000:         .WORD   ^M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,IV,DV>
                       52 FB AF 9E  0002:         MOVAB   .-3, R2
                50 00000000 0G 9E  0006:         MOVAB   $PDATA, R0
                          51 50 D0  000D:         MOVL    R0, R1
                   00000000 GG 16  0010:         JSB     BAS$INIT_R8
                                    0016:
```

(continued on next page)

```
LISTER$MAIN                 Listing Tester
Generated code                                                         3-AUG-1982 14:16:32  VAX-11 BASIC V2.0-0          Page   7
                                                                       3-AUG-1982 10:25:05  BAS$$DISK:[PARODI.BASIC]LISTER.BAS
              FC AD FD AF 9E  0016:  $L_10:  MOVAB   $L_10, -4(FP)
                              001B:  $S_2:
                              001B:  $S_3:
                              001B:  $S_4:
                              001B:  $S_5:
                              001B:  FIRST_LOOP:
              FC AD FD AF 9E  001B:          MOVAB   .-1, -4(FP)
                 77 AB 00 D0  0020:          MOVL    #0, INDEX(R11)
                    77 AB D7  0024:          DECL    INDEX(R11)
        0002 77 AB 01 05 F1  0027:          ACBL    #5, #1, INDEX(R11), $T_0030
                       36 11  002E:          BRB     $T_0066
              FC AD FD AF 9E  0030:  $T_0030:MOVAB   .-1, -4(FP)
                       00 DD  0035:  $S_6:   PUSHL   #0
           00000000 GG 01 FB  0037:          CALLS   #1, BAS$PRINT
     5C 00 01 05 00 77 AB 0A  003E:          INDEX   INDEX(R11), #0, #5, #1, #0, R12
              7E 5F AB 4C 50  0046:          MOVF    Q(R11)[R12], -(SP)
           00000000 GG 01 FB  004B:          CALLS   #1, BAS$OUT_F_V_B
           00000000 GG 00 FB  0052:          CALLS   #0, BAS$IO_END
                              0059:  $S_7:
              FC AD FD AF 9E  0059:  $T_0059:MOVAB   .-1, -4(FP)
              CD 77 AB 05 F3  005E:          AOBLEQ  #5, INDEX(R11), $T_0030
                    77 AB D7  0063:          DECL    INDEX(R11)
              FC AD FD AF 9E  0066:  $T_0066:MOVAB   .-1, -4(FP)
                              006B:  $S_8:
                              006B:  SECOND_LOOP:
              FC AD FD AF 9E  006B:          MOVAB   .-1, -4(FP)
              FC AD FD AF 9E  0070:  $T_0070:MOVAB   .-1, -4(FP)
              FFFFFFFF 8F D5  0075:          TSTL    #-1
                       18 13  007B:          BEQL    $T_0095
                 5C 77 AB 4E  007D:  $S_9:   CVTLF   INDEX(R11), R12
                    5C 08 40  0081:          ADDF2   #1., R12
                 77 AB 5C 4A  0084:          CVTFL   R12, INDEX(R11)
                 5C 77 AB 4E  0088:  $S_10:  CVTLF   INDEX(R11), R12
                    1A 5C 51  008C:          CMPF    R12, #5.
                       02 19  008F:          BLSS    $T_0093
                       02 11  0091:          BRB     $T_0095
                              0093:  $T_0093:
                       DB 11  0093:  $S_11:  BRB     $T_0070
              FC AD FD AF 9E  0095:  $T_0095:MOVAB   .-1, -4(FP)
                              009A:
                              009A:  $L_32767:
              FC AD FD AF 9E  009A:          MOVAB   $L_32767, -4(FP)
        50 00000000 0G 9E     009F:          MOVAB   $PDATA, R0
           00000000 GG 16     00A6:          JSB     BAS$END_R8
                    50 01 D0  00AC:          MOVL    #1, R0
                          04  00AF:          RET
                              00B0:          .END
```

1. This is the listing header.

   The first line contains: 1) the module name, 2) the text specified in the %TITLE directive, 3) the date and time of the compilation, 4) the version number of VAX–11 BASIC, and 5) the page number.

   The second line contains: 1) the text specified in the %IDENT directive, 2) the text specified in the %SBTTL directive, 3) the date and time the source file was created, and 4) the file specification of the source file.

2. These are statement numbers.

   This lists the number of the last statement on each line of text. BASIC uses these line and statement numbers when reporting compile time errors. Also, when using the VAX–11 Symbolic Debugger, these numbers help you set breakpoints on a multi-statement line.

3. This is %INCLUDE file information.

   The "I" tells you that this code was extracted from a %INCLUDE file. The number following the "I" tells you the depth of nested %INCLUDE directives. Because this %INCLUDE directive occurs in the source program, the number is "1". If the %INCLUDE file itself contained a %INCLUDE directive, the code extracted from that file would be numbered "2", and so on.

4. This is a true-false flag for %IF–%THEN–%ELSE–%END–%IF directives.

   Lines marked with "T" are compiled. Lines marked with "F" are not compiled.

5. This is the cross-reference listing for variables and named constants.

   This tells you the variable names, the line number and statement at which they are referenced, their data type, the PSECT containing them (if any), and their offset from the start of the PSECT.

6. This is the cross-reference listing for mapped variables.

   This tells you the variable names, the line number and statement number at which they are referenced, their data type, and their offset from the start of the MAP PSECT.

7. This is the label cross-reference listing for labels.

   This tells you the label names and the line number and statement at which they are referenced.

8. This is the allocation listing for the MAP named SHARED.

   This tells you the names of all variables in the MAP, their offset, in bytes, from the beginning of the MAP, their size in bytes, and their data type.

9. This section lists named constants.

   This tells you the names of all explicitly declared constants, their data-type, and the value assigned to them.

10. This is the allocation listing for dynamic variables.

    This part of the listing applies to variables that are neither parameters nor part of a COMMON or MAP PSECT. This tells you the names of the variables, their offset from R11, their size in bytes, and their data type.

11. This section lists the program sections (PSECTs).

    This tells you the names of all PSECTs, their size in bytes, and their attributes. See the *VAX–11 Linker Reference Manual* for more information on PSECT attributes.

12. This section lists all external references.

    This includes subprograms, external variables, constants, functions, and routines, and the RTL routines invoked to support BASIC language elements. For more information on these RTL routines, see the *VAX–11 Run-Time Library Language Support Procedures Reference Manual.*

13. This section lists the compiler defaults in effect when the program was compiled.

    For more information, see Section 2.1.24 in this manual.

14. This section lists the compiler-generated machine code.

    For a thorough understanding of this code, you should be an experienced VAX–11 MACRO programmer. Only the naming scheme for the compiler-generated labels is explained, as follows:

    - Symbols beginning with $L_n are line-number labels, where n is the line number.

    - Symbols beginning with $S_n are statement labels, where n is the statement number on a given line.

    - Symbols beginning with $T_n are compiler-generated labels, where n is the relative PC of the location.

    Note that these labels are used to improve the readability of the listing, and are not accessible from the VAX–11 Symbolic Debugger. See the *VAX–11 Run-Time Library Language Support Procedures Reference Manual* for more information on BAS$ and OTS$ routines.

Index

This index provides a complete cross-reference to the information in this manual. In the index the following conventions are used:

Example	Explanation
1–8t	A page number followed by a *t* indicates a table.
4–36f	A page number followed by an *f* indicates a figure.

For material not covered in this manual, see the Master Index in the back of the *BASIC Reference Manual*. The Master Index contains a list of the major references to information throughout the BASIC documentation set.

A

Accessing
  RECORD components, 6–6
  remote files, 8–13
Accounts, 1–2
ADDRESS qualifier
  of debugger commands, 4–15
Addresses
  defining symbolically, 4–14
  evaluating with debugger, 4–12
  examining with debugger, 4–10
  modifying with debugger, 4–11
  symbolic, 4–13
AFTER qualifier
  of SET BREAK command, 4–6
Allocating a tape drive, 8–2
ALLOW clause, 8–16
Ambiguous RECORD references, 6–4
ANSI D format, 8–3
ANSI F format, 8–3
ANSI format magnetic tapes, 8–4
ANSI Minimal BASIC, 7–1 to 7–6
  arrays in, 7–4
  built–in functions allowed, 7–3
  DEF functions, 7–3
  extensions to, 7–2
  implementation–defined features, 7–4
  input prompt, 7–5
  machine infinitesimal, 7–5
  machine infinity, 7–5
  margin for output line, 7–5
ANSI Minimal BASIC (Cont.)
  numeric constants, 7–2
  OPTION BASE statement, 7–4
  precision, 7–6
  print zone length, 7–6
  program format, 7–4
  random numbers, 7–6
  required statements, 7–2
  string length, 7–5
  variable initialization, 7–5
  variable names, 7–2
ANSI_STANDARD, 2–6, 2–20, 2–21, 7–1
APPEND, 2–4
Arc cosine, 5–20
Argument list, 3–8
Arrays
  as parameters, 3–11
  in CDD, 9–13
ASSIGN, 2–5
Assigning RECORD values, 6–7
AUDIT qualifier, 2–21, 9–2

B

BASE attribute
  in CDD, 9–8
BASIC
  compiler commands, 2–3t
  DCL command, 1–10, 1–11
  qualifiers of DCL command, 1–11
  using from DCL command level, 2–19

BASIC environment, 1–2 to 1–7, 2–1, 2–3
  compiling programs in, 1–4
  creating programs in, 1–3
  developing programs in, 1–3
  running programs in, 1–3
BASIC$LIB, 10–2
BIT data type
  in CDD, 9–12
BLOCKSIZE, 8–3, 8–5
BOUNDS, 2–6
  with /CHECK qualifier, 2–20, 2–22
BP2COMPATIBILITY
  with /FLAG qualifier, 2–20
Breakpoints
  cancelling, 4–6
  setting, 4–6
  showing, 4–6
BY DESC, 3–6
BY REF, 3–6
BY VALUE, 3–6
BYTE, 2–6

C

Calculator mode, 1–4
CALL, 3–4
Calling subprograms, 3–4
CANCEL BREAK debugger command, 4–5
CANCEL MODULE debugger command, 4–3
CANCEL SCOPE debugger command, 4–4
CANCEL TRACE debugger command, 4–6
CANCEL WATCH debugger command, 4–7
Card reader
  I/O from, 8–7
CDD, 9–1 to 9–13
CDD$DEFAULT, 9–2
CDD_DEFINITIONS
  with /SHOW qualifier, 2–21
CDDL, 9–3
Character strings
  CDD data types, 9–6
CHECK qualifier, 2–20, 2–22
CLOSE, 8–6
Commands
  compiler, 2–3 to 2–18
Common Data Dictionary (CDD), 9–1 to 9–13
  arrays in, 9–13
  BASE attribute, 9–8
  DIGITS attribute, 9–8
  FRACTION attribute, 9–8
  OCCURS clause, 9–13
  OCCURS DEPENDING clause, 9–13
  SIGNED integers, 9–8
  SIZE attribute, 9–8
Common Data Dictionary (CDD) (Cont.)
  UNSIGNED integers, 9–8
Communication
  task–to–task, 8–13
COMPILE, 1–4, 2–5
  command qualifiers, 2–6t, 2–6 to 2–7
Compiler commands, 2–3 to 2–18
Compiling programs
  from DCL level, 1–11
  in BASIC environment, 1–4
Compiling subprograms, 3–13
Complex numbers
  in CDD, 9–9
CONSTANT
  in EXTERNAL, 5–4
Constants
  symbolic, 5–2
CONTINUE, 2–7
Creating
  BASIC Programs, 2–1 to 2–18
  cross–reference lists, 2–22
  executable images, 1–11
  files with EDT, 1–8
  programs in BASIC environment, 1–3
  user–supplied libraries, 10–1
CRFSHR.EXE, 10–3
CROSS, 2–6, 2–20, 2–22
CTRL/Y debugger command, 4–10
Current location, 4–13

D

Data Definition Language Utility (CDDL), 9–3
Data structures, 6–1
Data types
  BASIC translations from CDD, 9–4t
  BIT in CDD, 9–12
  CDD character strings, 9–6
  common to BASIC and CDD, 9–4
  common to the CDD and BASIC, 9–4t
  DATE in CDD, 9–12
  decimal string in CDD, 9–11
  floating–point in CDD, 9–9
  in debugger, 4–15
  integers in CDD, 9–7
  setting default, 2–24, 2–25
  translation between CDD and BASIC, 9–4
  VIRTUAL in CDD, 9–12
DATE data type
  in CDD, 9–12
DCL command level
  using BASIC from, 2–19
DCL commands
  ASSIGN, 8–2

DCL commands (Cont.)
BASIC, 1–10, 1–11
DELETE, 1–10
DIRECTORY, 1–9
LINK, 1–12
PRINT, 1–10
TYPE, 1–9
using from within BASIC, 2–5
DCL LIBRARY command, 10–1
DEBUG qualifier
of COMPILE command, 2–6, 4–1
of DCL BASIC command, 2–20, 2–23, 4–1
of DCL LINK command, 4–1
Debugger, 4–1 to 4–15
calling subprograms from, 4–14
command qualifiers, 4–14, 4–15t
commands and keywords, 4–2t
keywords, 4–2
symbol table, 4–2
Debugging
in immediate mode, 1–6
with symbolic debugger, 4–1 to 4–15
Decimal string data types, 9–11
DECIMAL_SIZE, 2–20, 2–23
Declaring
subprograms, 3–3
symbolic constants, 5–5
system services, 5–4
DECLINING
with /FLAG qualifier, 2–20
Defaults
compiler options, 2–28
for file specification, 1–8
setting data type, 2–24, 2–25
DELETE, 2–8
DCL command, 1–10
Deleting
files, 1–10
program lines, 2–8
DEPOSIT debugger command, 4–11
Depositing values
with debugger, 4–11
Developing programs, 1–1
Device, 1–8
unit record, 8–7
Device–specific I/O, 8–7 to 8–11
closing magnetic tape files, 8–9
opening disk files, 8–10
opening magnetic tapes, 8–7
reading from disk files, 8–11
reading from magnetic tapes, 8–8
RESTORE, 8–9
to disks, 8–9 to 8–11
to magnetic tape, 8–7 to 8–9
Device–specific I/O (Cont.)
writing to disk files, 8–11
writing to magnetic tapes, 8–8
DIGITS attribute
in CDD, 9–8
DIRECTORY
DCL command, 1–9
Directory, 1–8
Displaying files, 1–9
DO qualifier
of SET BREAK debugger command, 4–6
DOUBLE, 2–6, 2–20

E

EDIT, 2–8
Editing programs
with EDIT command, 2–8
Editor
text, 1–8
EDT, 1–8
creating files with, 1–8
sample session, 1–9
Elementary RECORD component, 6–2
Elliptical references, 6–6
rules for, 6–6
END FUNCTION, 3–12
END GROUP, 6–2
END RECORD, 6–1
END SUB, 3–9
ENVIRONMENT
with /SHOW qualifier, 2–21
EVALUATE debugger command, 4–12
Evaluating
addresses with debugger, 4–12
expressions with debugger, 4–12
locations with debugger, 4–12
EXAMINE debugger command, 4–10
Examining
addresses with debugger, 4–10
locations with debugger, 4–10
Examples
of I/O to ANSI magnetic tape, 8–17 to 8–22
of system services, 5–4 to 5–14
Executable images
creating, 1–11
EXIT, 2–9
EXIT debugger command, 4–10
EXIT FUNCTION, 3–12
EXIT SUB, 3–9
EXPLICIT
with /TYPE_DEFAULT, 2–21
Explicit record locking, 8–16

Expressions
  evaluating with debugger, 4–12
EXTERNAL
  declaring symbolic constants, 5–4
  declaring system services, 5–4
  subprogram declarations, 3–3
External names
  resolving, 5–14

F

File name, 1–8
File sharing, 8–15
File specifications, 1–7
  defaults, 1–8t
  using logical names, 8–2
File type, 1–8
Files
  deleting, 1–10
  displaying with DIRECTORY, 1–9
  displaying with TYPE, 1–9
  printing, 1–10
  specifying, 1–7
  version number, 1–10
FIND
  with ALLOW clause, 8–16
FIXED, 8–3
Fixed–point data types
  in CDD, 9–7
FLAG, 2–20, 2–24
Floating–point data types
  in CDD, 9–9
FRACTION attribute
  in CDD, 9–8
FUNCTION, 3–12
  in EXTERNAL, 5–4
FUNCTION subprograms
  ending, 3–12
  exiting, 3–12

G

GET
  to ANSI format magnetic tapes, 8–5
  with ALLOW clause, 8–16
  with REGARDLESS clause, 8–16
GFLOAT, 2–6
GO debugger command, 4–8
GROUP, 6–2
  block, 6–2
Grouping RECORD components, 6–2 to 6–3

H

HELP, 2–9
HEX qualifier
  of debugger commands, 4–15
HFLOAT, 2–6

I

I/O
  device–specific, 8–7 to 8–11
  from card reader, 8–7
  network, 8–13
  on VAX/VMS, 8–2 to 8–22
  queueing requests, 5–10
  to magnetic tape (RMS), 8–2
  to mailboxes, 8–12
IDENTIFY, 2–11
Images
  shareable, 10–4 to 10–5
Immediate mode, 1–4
  debugging in, 1–6
INCLUDE
  with /SHOW qualifier, 2–21
%INCLUDE %FROM %CDD, 9–2
Instance
  of a RECORD, 6–2
Integer data types
  in CDD, 9–7
INTEGER_SIZE, 2–20, 2–24
Integers
  SIGNED in CDD, 9–8
  UNSIGNED in CDD, 9–8

J

Job information, 5–12

L

LBRSHR.EXE, 10–3
LEFT OVERPUNCHED NUMERIC, 9–11
LEFT SEPARATE NUMERIC, 9–11
LIB$FREE_LUN, 5–18
LIB$FREE_TIMER, 5–16
LIB$GET_LUN, 5–18
LIB$INIT_TIMER, 5–16
LIB$STAT_TIMER, 5–16
Libraries, 10–1 to 10–5
  searched by BASIC, 10–2
  searched by VAX–11 Linker, 10–2
  user–supplied, 10–1
%LINE, 4–6, 4–7
LINES, 2–6, 2–20, 2–24

LINK
  DCL command, 1–12
Linking
  programs, 1–11
  with user–supplied libraries, 10–1
LIST, 2–7, 2–11, 2–20, 2–24
Listing
  creating a, 2–24
  cross–reference, 2–22
LISTNH, 2–11
Lists
  traceback, 4–1
LOAD, 2–11
Local copies, 3–8
Locations
  current, 4–13
  evaluating with debugger, 4–12
  examining with debugger, 4–10
  modifying with debugger, 4–11
  next, 4–13
  previous, 4–13
LOCK, 2–12
Locking records
  explicitly, 8–16
Logging in, 1–2
Logical names, 8–2
  translating, 5–6
Logical unit numbers, 5–18
LONG, 2–7, 2–20
LUN, 5–18

M

MACHINE, 2–7, 2–20, 2–25
  with /SHOW qualifier, 2–21
Magnetic tape
  allocating for device–specific I/O, 8–7
  ANSI D format, 8–3
  ANSI F format, 8–3
  ANSI format, 8–4
  block size, 8–3
  mounting, 8–2
  opening FOR INPUT, 8–3
  opening FOR OUTPUT, 8–3
  positioning, 8–4
  reading from ANSI format, 8–5
  RMS I/O to, 8–2
  writing to ANSI format, 8–4
Magnetic tapes
  device–specific I/O to, 8–7 to 8–9
  rewinding, 8–6
Mailboxes, 5–5, 8–12
MAP
  with /SHOW qualifier, 2–21
Masks, 5–3
Measuring performance, 5–16
Modifiable parameters, 3–8
Modifying
  addresses with debugger, 4–11
  locations with debugger, 4–11
Module
  cancelling with debugger, 4–4
  setting with debugger, 4–3
  showing with debugger, 4–3
MTH$ACOS, 5–20
Multiple program units
  debugging, 2–29
  running, 2–29

N

Named constants
  evaluating with debugger, 4–15
Network I/O, 8–13
NEW, 2–12
Next location, 4–13
Node, 1–8
NOREWIND, 8–4
NOTRACE, 4–1
Null parameters, 3–8
Numbers, complex
  in CDD, 9–9

O

OBJECT, 2–7, 2–20, 2–25
Object modules
  loading with LOAD, 2–11
OCCURS clause
  in CDD, 9–13
OCCURS DEPENDING clause
  in CDD, 9–13
OCT qualifier
  of debugger commands, 4–15
OLD, 1–4, 2–13
OPEN, 8–3
OPTION BASE, 7–4
OVERFLOW, 2–7
  with /CHECK qualifier, 2–20, 2–22
OVERRIDE
  with /SHOW qualifier, 2–21

P

PACKED NUMERIC, 9–11
Parameter passing mechanisms, 3–6
Parameters
  argument lists, 3–8

Parameters (Cont.)
  arrays, 3–11
  local copies, 3–8
  modifiable, 3–8
  null, 3–8
  passing mechanisms, 3–6
Passing mechanisms
  for parameters, 3–6
Password, 1–2
Path name
  RECORD component, 6–4
Performance measuring, 5–16
Positioning magnetic tape, 8–4
Previous location, 4–13
PRINT
  DCL command, 1–10
Printing files, 1–10
Process information, 5–12
Program development
  in BASIC environment, 1–3
  methods, 1–1, 2–1
Program development methods, 2–2f
Program execution
  continuing, 2–7
  controlling with debugger, 4–5
Program lines
  deleting, 2–8
Program listing
  creating a, 2–24
  cross–reference, 2–22
Program segmentation, 3–1 to 3–16
  techniques, 3–2
Programs
  debugging, 4–1 to 4–15
  editing with EDIT command, 2–8
PUT
  to ANSI format magnetic tapes, 8–4

Q

Qualifiers, 2–19 to 2–28
  default, 2–28
  examples of, 2–19
  negation of, 2–19
  of COMPILE command, 2–6 to 2–7
  of DCL BASIC command, 1–11
  of debugger commands, 4–14
  of the DCL BASIC command, 2–19 to 2–28, 2–20t
  specifying, 2–19
Qualifying RECORD components, 6–4

R

REAL_SIZE, 2–20, 2–25
RECORD
  accessing components, 6–6
  ambiguous references, 6–4
  assignment, 6–7
  block, 6–2
  elementary components, 6–2
  fully qualified components, 6–4
  grouping components, 6–2 to 6–3
  instance, 6–2
  statement, 6–1 to 6–8
  template, 6–2
  used with CDD, 9–3
  variants, 6–5
Record locking, 8–15
Records
  blocking on magnetic tape, 8–6
  locking explicitly, 8–16
RECORDSIZE, 8–5
References
  ambiguous RECORD, 6–4
  elliptical, 6–6
  to RECORD components, 6–6
REGARDLESS clause, 8–16
Remote file access, 8–13
RENAME, 2–13
REPLACE, 2–13
RESEQUENCE, 2–14
Resolving external names, 5–14
RESTORE
  for magnetic tape files, 8–6
Return status, 5–2
Rewinding magnetic tapes, 8–6
RIGHT OVERPUNCHED NUMERIC, 9–11
RIGHT SEPARATE NUMERIC, 9–11
ROUND, 2–7, 2–26
Routine
  definition of, 3–2
RTL routines, 5–14 to 5–20
  types of, 5–15
RUN, 1–4, 2–14
Run–Time Library, 5–14 to 5–20
RUNNH, 2–14
Running programs
  in BASIC environment, 1–3

S

SAVE, 2–15
SCALE, 2–15, 2–21, 2–26
SCOPE, 4–13

Scope
cancelling with debugger, 4–4
setting with debugger, 4–4
showing with debugger, 4–4
specifying, 4–13
SCRATCH, 2–15
SCRSHR.EXE, 10–3
SEQUENCE, 2–15
SET, 2–16
SET BREAK debugger command, 4–5
SET LANGUAGE debugger command, 4–3
SET MODULE debugger command, 4–3
SET SCOPE debugger command, 4–4
SET TRACE debugger command, 4–6
SET WATCH debugger command, 4–7
SETUP, 2–7
Severity level, 5–2
(See also Status codes)
Shareable images, 10–4 to 10–5
accessing, 10–4
creating, 10–4
Sharing data
among subprograms, 3–10
Sharing files, 8–15
SHOW, 2–16, 2–21, 2–26
SHOW BREAK debugger command, 4–5
SHOW CALLS debugger command, 4–8
SHOW LANGUAGE debugger command, 4–3
SHOW MODULE debugger command, 4–3
SHOW qualifier
CDD_DEFINITIONS, 9–1
SHOW SCOPE debugger command, 4–4
SHOW TRACE debugger command, 4–6
SHOW WATCH debugger command, 4–7
SIGNED integers, 9–8
SIGNED NUMERIC, 9–11
SINGLE, 2–7, 2–21
SIZE attribute
in CDD, 9–8
SOURCE
with /SHOW qualifier, 2–21
Specifying SCOPE, 4–13
STARLET.MLB, 10–3
STARLET.OLB, 10–3
Statements
line–numbered, 2–3
Status codes
for system services, 5–2
STEP debugger command, 4–8
SUB, 3–9
Subprograms, 3–1 to 3–16
BASIC, 3–8 to 3–16
called by other languages, 3–15
calling, 3–4
Subprograms (Cont.)
calling from debugger, 4–14
compiling, 3–13
declaring, 3–3
ending SUBs, 3–9
exiting from SUBs, 3–9
FUNCTION, 3–12 to 3–16
non–BASIC, 3–14
passing arrays to, 3–11
sharing data, 3–10
SUB, 3–9 to 3–12
Symbol table
debugger, 4–2
Symbolic addresses, 4–13
defining, 4–14
Symbolic constants, 5–2
declaring, 5–5
Symbolic debugger, 4–1 to 4–15
SYMBOLS
with /DEBUG qualifier, 2–20, 2–23
Syntax checking
line–by–line, 2–27
SYNTAX_CHECK, 2–7, 2–21, 2–27
SYS$CREMBX, 5–5, 8–12
SYS$GETJPI, 5–12
SYS$GETMSG, 5–7
SYS$QIOW, 5–10
SYS$TRNLOG, 5–6
System service examples, 5–4 to 5–14
System services, 5–2 to 5–14
declaring, 5–4
examples, 5–4 to 5–14
status codes, 5–2

T

Task–to–task communication, 8–13
Template
of a RECORD, 6–2
Text editor, 1–8
TRACEBACK, 2–7
with /DEBUG qualifier, 2–20, 2–23
Traceback lists, 4–1
Tracepoints
cancelling, 4–7
setting, 4–7
showing, 4–7
Translating
logical names, 5–6
TYPE
DCL command, 1–9
TYPE_DEFAULT, 2–21, 2–27

U

Unit record devices, 8–7
UNLOCK EXPLICIT clause, 8–16
UNSAVE, 2–18
UNSIGNED integers, 9–8
UNSIGNED NUMERIC, 9–11
Username, 1–2
Using BASIC
  from DCL command level, 1–10

V

Values
  depositing with debugger, 4–11
VARIANT, 2–7, 2–21, 2–27
Variants
  of a RECORD, 6–5
Version, 1–8
Version number, 1–10
VIRTUAL data type
  in CDD, 9–12
VMSRTL.EXE, 10–3

W

WARNING, 2–7, 2–27
Watchpoints
  cancelling, 4–8
  setting, 4–8
  showing, 4–8
WORD, 2–7, 2–21

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**

In Canada call **800–267–6146**

In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary:
809–754–7575

BASIC on
VAX/VMS Systems
AA–L336A–TK

Reader's Comments

Note: This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.________________________________

Did you find errors in this manual? If so, specify the error and the page number.________________________________

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)________________________________

Name________________________________ Date________________

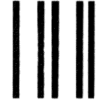
Organization________________________________

Street________________________________

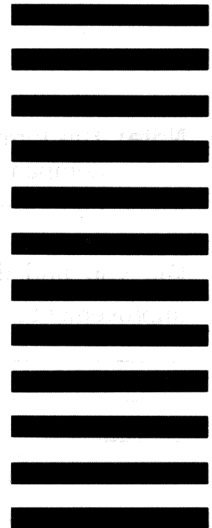
City________________________________ State________________ Zip Code or Country________________

Do Not Tear - Fold Here and Tape

digital



No Postage
Necessary
if Mailed in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK01–2/H03
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK, N.H. 03054

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line