VAX Ada

digital

Developing Ada Programs on VMS Systems

Order Number: AA–EF86B–TE

Developing Ada Programs on VMS Systems

Order Number: AA–EF86B–TE

May 1989

This manual describes how to compile, link, execute, and debug VAX Ada programs. It describes the use of the VAX Ada compiler, VAX Ada program library manager, and VMS Debugger.

Revision/Update Information: This revised manual supersedes *Developing Ada Programs on VAX/VMS* (Order No. AA–EF86A–TE)

Operating System and Version: VMS Version 5.0 or higher

Software Version: VAX Ada Version 2.0

digital equipment corporation
maynard, massachusetts

February 1985
Revised, May 1989

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

ALL–IN–1	EduSystem	RT
DEC	IAS	ULTRIX
DEC/CMS	MASSBUS	UNIBUS
DEC/MMS	PDP	VAX
DECmate	PDT	VAXcluster
DECnet	P/OS	VAXELN
DECsystem–10	Professional	VMS
DECSYSTEM–20	Q–bus	VT
DECUS	Rainbow	Work Processor
DECwriter	RSTS	
DIBOL	RSX	digital™

ZK3294

Contents

Appendix E Efficient Compilation

Appendix F Compile-Time Diagnostic Messages

Tables

Preface

This manual describes how to compile, link, execute, and debug VAX Ada programs. It describes the use of the VAX Ada compiler, VAX Ada program library manager, and VMS Debugger.

Intended Audience

This manual is intended for any programmer who needs information on compiling, linking, executing, and debugging VAX Ada programs. The reader should have a working knowledge of Ada, the Digital Command Language (DCL), and DCL command procedures. Experience with compiling and linking another VMS-supported language is helpful.

Structure of This Document

This manual has seven chapters and nine appendixes. The first chapter provides introductory material on VAX Ada and the VAX Ada programming environment. The remaining chapters discuss in detail how to compile, link, execute, and debug VAX Ada programs and use the VAX Ada program library manager.

The appendixes provide reference information on ACS commands, debugger commands, using VAX Ada with the VAX Language-Sensitive Editor (LSE) and VAX Source Code Analyzer (SCA), memory usage, diagnostic messages, and problem reporting.

Associated Documents

For more information on the VAX Ada language, see the *VAX Ada Language Reference Manual*; for more information on implementation details of VAX Ada in the context of the VMS operating system, see the *VAX Ada Run-Time Reference Manual*. You should have all or most of the VMS system documentation available for reference.

The following Ada textbooks may be of interest:

- Barnes, J.G.P. *Programming in Ada*. Reading, Massachusetts: Addison-Wesley, second edition, 1984.
- Booch, Grady. *Software Components with Ada: Structures, Tools and Subsystems*. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., 1987.
- Booch, Grady. *Software Engineering with Ada*. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., second edition, 1987.
- Cherry, G.W. *Parallel Programming in ANSI Standard Ada*. Reston, Virginia: Reston Publishing Company, Inc., 1984.
- Gehani, Narain. *Ada, Concurrent Programming*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1984.
- Habermann, A.N., and D.E. Perry. *Ada for the Experienced Programmer*. Reading, Massachusetts: Addison-Wesley, 1983.
- Weiner, Richard, and Richard Sincovec. *Programming in Ada*. New York: John Wiley & Sons, 1983.

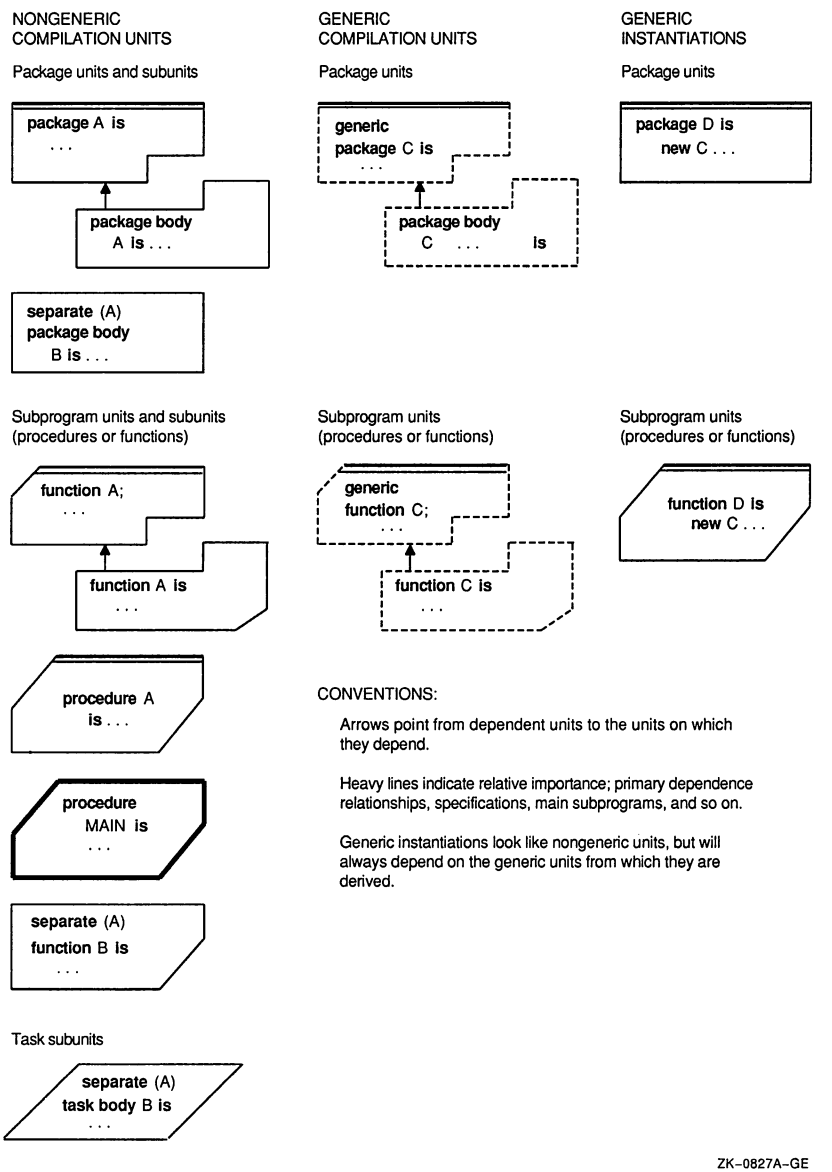
Conventions

Convention	Meaning
RETURN	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, RETURN.
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.

Convention	Meaning
$ SHOW TIME 05-JUN-1988 11:55:22	Interactive examples show all output lines or prompting characters that the system prints or displays in black. All user-entered commands are shown in red.
. . .	A horizontal ellipsis in an Ada example or figure indicates that not all of the statements are shown.
. . .	A vertical ellipsis in an interactive figure or example indicates that not all of the commands and responses are shown.
file-spec, . . .	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates that additional parameters, options, or values can be entered.
task	Boldface indicates Ada reserved words.
*type*_name	Italicized words in syntax descriptions indicate descriptive prefixes that are intended to give additional semantic information rather than to define a separate syntactic category.
[expression]	Square brackets indicate that the enclosed item is optional. (However, square brackets are not optional in the syntax of a directory name in a file specification.)
{, mechanism_name }	Braces in Ada syntax indicate that the enclosed item can be repeated zero or more times. Braces in debugger command syntax enclose lists from which you must choose one item.
quotation marks apostrophes	The term "quotation marks" is used to refer to double quotation marks ("). The term "apostrophe" is used to refer to a single quotation mark (').

Figure 1 explains the shapes and conventions used in figures that diagram Ada programs.

Figure 1: Figure Conventions



New and Changed Features

For this release, this manual has been reorganized, information has been clarified and corrected, and examples have been added.

This version of the manual also discusses the following features, which have been added or changed since VAX Ada Version 1.0:

- By default, the completion of a generic instantiation no longer causes units that depend on the unit containing the incomplete instantiation to become obsolete. (You can revert to the Version 1 behavior by using the pragma INLINE_GENERIC or by specifying the equivalent /OPTIMIZE qualifier options at compilation time.)
- The ACS COMPILE command has changed; when recompiling obsolete units or completing incomplete generic units it now looks for external source files first, and uses copied source files only when external source files are not available.
- The default job name for the ACS COMPILE, LINK/NOMAIN, and RECOMPILE commands has changed. If you have not specified a job name with the /NAME qualifier to these commands, the program library manager creates a name comprising up to the first 39 characters of the first unit specified in the command. Previously, the program library manager created a name using the first unit compiled.

  If the first unit specified contains a wildcard character, then the default name for the job is ACS_COMPILE, ACS_LINK, or ACS_RECOMPILE, as appropriate.

  This new behavior also applies to the new ACS LOAD command; however, because the ACS LOAD command accepts file names, not unit names, the program library manager creates a name from the first file specified in the command.

- The ACS COPY UNIT, ENTER UNIT, DELETE UNIT, and MERGE commands apply only to the specification and body of the specified units; they no longer automatically apply to subunits of the specified units.
- The ACS MERGE command now checks for a more recent external source file for a unit in the parent library with the same name as a unit in the sublibrary. This check is done to prevent units in the parent library from being overwritten by units from the sublibrary if the parent units come from more recent external source files.
- When you specify an invalid directory specification with the ACS SET LIBRARY command, the program library manager now sets the library to whatever you specified. This effect prevents you from incorrectly modifying the wrong library.
- The output from the ACS SHOW LIBRARY/FULL and SHOW PROGRAM commands has been enhanced.
- An ACS LOAD command has been added. It processes units in the specified source files, and puts them into the current program library as obsolete units. This command is useful for putting a set of units into a library for the first time, especially if you do not know the compilation order. It is also useful for adding units to an existing program.
- The ACS CONVERT LIBRARY, REORGANIZE, and SHOW VERSION commands have been added.
- A /BATCH_LOG qualifier has been added to the ACS COMPILE, RECOMPILE, and LINK commands. The new ACS LOAD command also has a /BATCH_LOG qualifier.
- The /BODY qualifier to the ACS COMPILE and RECOMPILE commands has been deleted. In its place is a new /FORCE_BODY qualifier, which provides the capability of the former combination of /BODY and /NODATE_CHECK qualifiers.
- The /BODY qualifier to the ACS DELETE UNIT command has been renamed /BODY_ONLY. A similar /BODY_ONLY qualifier has been added to the ACS COPY UNIT, DIRECTORY, ENTER UNIT, EXTRACT SOURCE, MERGE, REENTER, and SHOW LIBRARY/UNITS commands.
- The behavior of the /COMMAND qualifier to the ACS COMPILE, RECOMPILE, or LINK/NOMAIN command has been corrected and changed. If you do not specify a file name with this qualifer, the program library manager creates a name comprising up to the first 39 characters of the first unit specified in the command. Previously, the program library manager created a name using the name of the first unit compiled.

If the first unit specified contains a wildcard character, then the default name for the command file is ACS_COMPILE, ACS_RECOMPILE, or ACS_LINK, as appropriate

This new behavior also applies to the /COMMAND qualifier to the new ACS LOAD command.

- The /[NO]ENTERED qualifier to the ACS DIRECTORY command now allows you to optionally specify a program library. A similar /[NO]ENTERED qualifier has been added to the ACS COPY UNIT, DELETE UNIT, ENTER UNIT, EXTRACT SOURCE, MERGE, and SHOW LIBRARY/UNITS commands. A similar /ENTERED qualifier has been added to the ACS REENTER command.
- The behavior of the /[NO]ERROR_LIMIT qualifier to the DCL ADA and ACS COMPILE and RECOMPILE commands has changed. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues. Previously, the first unit that reached the error limit caused the entire compilation to terminate.

  This new behavior also applies to the /[NO]ERROR_LIMIT qualifier to the new ACS LOAD command.
- A /LOAD qualifier has been added to the DCL ADA command.
- A /[NO]LOCAL qualifier has been added to the ACS COPY UNIT, DELETE UNIT, DIRECTORY, ENTER UNIT, EXTRACT SOURCE, MERGE, and SHOW LIBRARY/UNITS commands. It controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are selected for the given operation. Note that when you specify the /LOCAL qualifier, entered units are selected unless the /NOENTER qualifier is also in effect (the defaults for these commands are /LOCAL and /ENTERED).
- An /INCLUDE qualifier has been added to the ACS LINK command.
- The behavior of the /OPTIMIZE qualifier for the compilation commands has changed, and a new set of qualifier options has been added (DEVELOPMENT, INLINE, SHARE, and so on). These options allow you to control subprogram and generic inline expansion, as well as generic code sharing, in the code generated for your program by the compiler.
- An /OBJECT qualifier has been added to the ACS ENTER FOREIGN command.
- A /PRELOAD qualifier has been added to the ACS COMPILE and RECOMPILE commands.

- The /SHOW=PORTABILITY option is the default when you specify the /LIST qualifier with the DCL ADA and ACS COMPILE and RECOMPILE commands.
- A /SPECIFICATION_ONLY qualifier has been added to the ACS COMPILE, COPY UNIT, DELETE UNIT, DIRECTORY, ENTER UNIT, EXTRACT SOURCE, MERGE, RECOMPILE, and REENTER commands.
- The behavior of the /SYNTAX_ONLY qualifier to the DCL ADA command has changed. It now updates the program library with successfully syntax-checked units. Previously, it processed the specified files, but did not update the library. You can prevent the /SYNTAX_ONLY qualifier from updating the library by also specifying the new /NOLOAD qualifier.
- A /USERLIBRARY qualifier has been added to the ACS LINK command.
- When an ACS COMPILE/WAIT, RECOMPILE/WAIT, LOAD/WAIT, LINK, or LINK/WAIT command is entered, the subprocess generated to execute the compiler or linker command file inherits all process logical names.
- Program libraries can be accessed across DECnet.
- Error messages have been been revised and improved.
- Messages about incomplete generic units are now status-level messages. To see them during compilation, you need to use the /WARNINGS=(STATUS:TERMINAL) qualifier on your compilation command.
- By default, the debugger obtains the displayed Ada source code from either copied source files or external source files, depending on whether or not the units were compiled with the /COPY_SOURCE qualifier (the /COPY_SOURCE qualifier is the default for all of the VAX Ada compilation commands).
- The debugger uses Ada unit-name rather than file-name conventions in symbol names and path names. However, specifications are still distinguished from bodies by an appended underscore.
- The debugger SHOW TASK/STATISTICS command no longer reports statistics on locks.
- Support for the VAX Source Code Analyzer (SCA) has been added. An /ANALYSIS_DATA qualifier can be specified with any of the compilation commands (DCL ADA and ACS COMPILE and RECOMPILE) to produce data analysis files for SCA. A predefined library of data analysis files is provided for the VAX Ada predefined units (the units in the library ADA$PREDEFINED); this library has the logical name ADA$SCA_PREDEFINED.

- The built-in predefined units (package STANDARD, package SYSTEM, procedure UNCHECKED_DEALLOCATION, function UNCHECKED_CONVERSION, and so on) are visible units in the library of predefined units (ADA$PREDEFINED).
- VAXELN is a possible system choice for the /SYSTEM_NAME qualifiers to the ACS CREATE LIBRARY, CREATE SUBLIBRARY, EXPORT, LINK, and SET PRAGMA commands.
- The package VAXELN_SERVICES is part of the set of VAX Ada predefined packages, and is a visible unit in the library of predefined units (ADA$PREDEFINED).

Chapter 1

Introduction to the VAX Ada Program Development Environment

The Ada programming language is a general-purpose, block-structured language. Ada is strongly typed, supports the abstraction of data and algorithms, promotes modular programming, provides for exact or approximate numerical calculations, and supports concurrency. Thus, Ada is well-suited for writing and maintaining large, complex systems that may include real-time or concurrent operations.

VAX Ada implements the American National Standards Institute (ANSI) and International Standards Organization (ISO) standard Ada programming language on the VMS operating system. Where allowed by the standard, VAX Ada also implements features designed to make programming in the VMS environment convenient and efficient.

The environment for developing VAX Ada programs consists of the set of tools and utilities provided by VAX Ada and the VMS operating system, plus any optional layered products you have installed on your system.

VAX Ada provides a program library manager, which is also the user interface to the VAX Ada compiler and the VMS Linker (linker). The VMS operating system provides the VMS Debugger (debugger) and a choice of text editors. Some of the optional layered products that you can install for use in developing VAX Ada programs are:

- The VAX Language-Sensitive Editor (LSE)
- The VAX DEC/Code Management System (CMS)
- The VAX DEC/Test Manager
- The VAX Performance and Coverage Analyzer (PCA)
- The VAX Source Code Analyzer (SCA)
- Various VAX Information Architecture products

VAX Ada is an integral part of the development environment for VAXELN Ada, which allows Ada programs to be developed on a VMS system and run on a VAXELN target. VAX Ada is also related to XD Ada, a family of VMS cross-compilers that produce Ada code for a number of non-VAX targets. See the *VAXELN Ada User's Manual* for more information on VAXELN Ada. See the XD Ada documentation for more information on XD Ada.

This chapter provides a step-by-step tutorial on developing Ada programs. This chapter also provides an overview of the program library manager and its command language, and explains the VAX Ada conventions and terminology related to compiling, linking, and managing program libraries.

1.1 Getting Started with VAX Ada

When you develop a VAX Ada program, you perform the following steps:

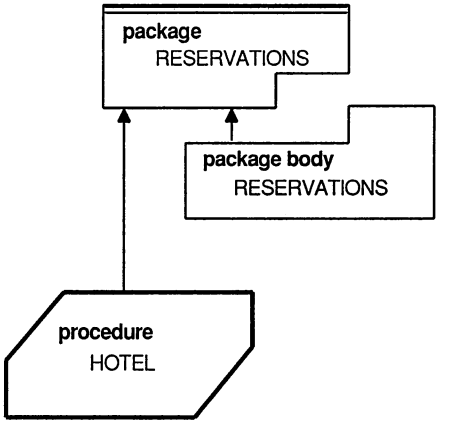
1. Create a working directory for your Ada source files, and define a current default directory for operations such as editing, debugging, and so on.
2. Create Ada source files for all of the compilation units in your program.
3. Create a program library.
4. Define a current program library for operations such as compilation, recompilation, and so on.
5. Compile the program into the current program library.
6. Link the program.
7. Execute the program.
8. Debug the program, if necessary.
9. Go back to step 5, and compile the program again if debugging has resulted in modifications to any of the source files.

The following sections explain these steps using an example program. The program, a hotel reservation system, consists of a *main program* named HOTEL and a *library package* named RESERVATIONS. The program has three compilation units:

- The specification of the library package RESERVATIONS
- The body of the library package RESERVATIONS
- The procedure body HOTEL, which names the library package RESERVATIONS in a **with** clause

Figure 1–1 shows the dependences among these compilation units. The dependences affect the order in which the compilation units can be compiled, and determine how the units must be recompiled as units are modified, compiled again, and so on. In Figure 1–1, arrows point from dependent units to the units they depend on.

Figure 1–1: Dependences Among the Hotel Reservation Program Compilation Units



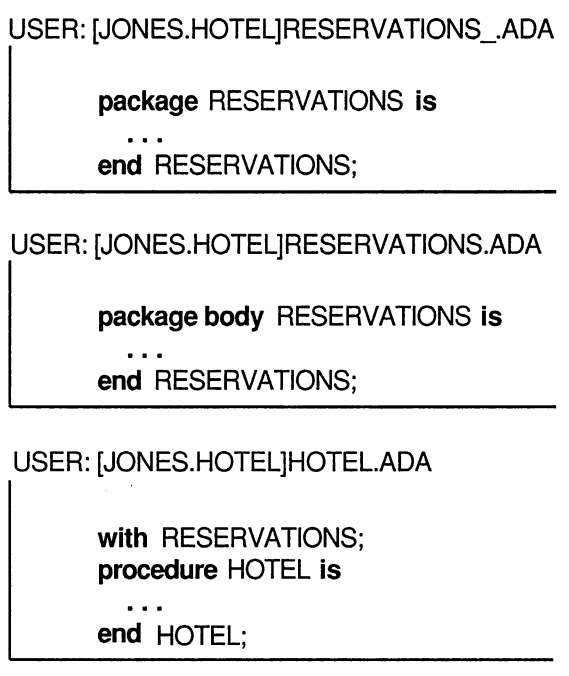
ZK–6743–GE

Figure 1–2 shows the source files and the relevant fragments of the compilation units for the example program. Note the following points:

- Each compilation unit is in a separate source file.
- The name of each source file matches the name of the compilation unit it contains. Specifications and bodies share the same unit name. However, the name of the source file for the package specification has a trailing underscore character (RESERVATIONS_.ADA) to distinguish it from the source file for the package body (RESERVATIONS.ADA). (These file-name conventions are also used by the VAX Ada program library manager and the VMS Debugger.)

- The working directory and current default directory are the VMS directory [JONES.HOTEL].
- The program library is the VMS directory [JONES.HOTEL.ADALIB] (in this case a subdirectory of the working directory and current default directory).

Figure 1–2: Source Files for the Hotel Reservation Program



1.1.1 Creating a Working Directory and Defining a Current Default Directory

The first steps in developing a VAX Ada program are to create a *working directory* and define a *current default directory*. You create a working directory by entering the DCL CREATE/DIRECTORY command. You define a current default directory by entering the DCL SET DEFAULT command. For example:

```
$ CREATE/DIRECTORY [JONES.HOTEL]
$ SET DEFAULT [JONES.HOTEL]
```

The working directory is the directory that contains your source files; the current default directory is the target directory for DCL commands (such as text-editing commands) and for some of the files produced during program development. As shown in the preceding example, these directories are usually the same.

1.1.2 Creating a Source File

You create an Ada source file in your working directory by using a text editor. For example:

```
$ EDIT HOTEL.ADA
```

This command invokes VAX EDT, the VMS default editor. EDT is one of two interactive text editors available with the VMS operating system. The other VMS editor is the Extensible VAX Editor (EVE), which is an interface to the VAX Text Processing Utility (VAXTPU).

You can also use the VAX Language-Sensitive Editor (LSE) to create Ada source files. LSE is an optional, multilanguage text editor designed specifically for software development. LSE provides formatted language templates to help you construct syntactically correct Ada source code, and allows you to compile, review, and correct compilation errors from within the editor. VAXTPU is part of and accessible from LSE. LSE is integrated with the VAX Source Code Analyzer (SCA) and VAX DEC/Code Management System (CMS). See Appendix C for Ada-specific information on LSE and for an example of using LSE to develop an Ada program.

For further information on the available text editors, see the following manuals:

- *Guide to VMS Text Processing*—provides tutorial information on the EDT editor, EVE editor, and Digital Standard Runoff (DSR)

- *VAX EDT Reference Manual*—provides comprehensive reference information on the EDT editor
- *VAX Text Processing Utility Manual*—provides comprehensive reference information on VAXTPU and EVE
- *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*—provides tutorial and reference information on LSE

1.1.3 Creating a Program Library

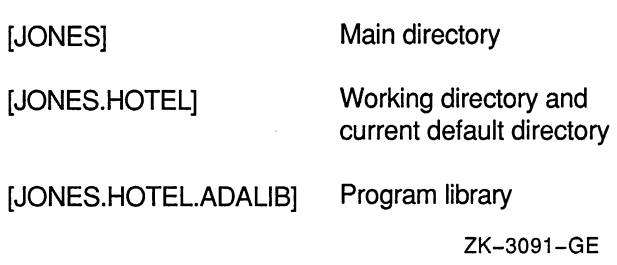
To compile or link an Ada program, you must have a *program library*. A program library is a special VMS directory that you create with the ACS CREATE LIBRARY command, specifying the name of the directory as a parameter. For example:

```
$ ACS CREATE LIBRARY [JONES.HOTEL.ADALIB]
```

The program library holds the products of VAX Ada compilations (object files and so on), and is used by the program library manager to keep track of compilation units. You should not use it for any purpose other than the one for which it was designed; for example, do not use it to store Ada source files or other files that have not been created by the program library manager.

Figure 1–3 shows the directory structure for the hotel reservation program. In this case, the program library [JONES.HOTEL.ADALIB] is a subdirectory of the working directory that contains the source files.

Figure 1–3: Directory Structure for the Hotel Reservation Program



VAX Ada also allows you to create one or more *program sublibraries*. See Chapters 2 and 5 for more information on using sublibraries.

1.1.4 Defining the Current Program Library

To use a program library for a compilation, you must first define it as the *current program library*. You define a current program library by entering the ACS SET LIBRARY command, specifying the name of the program library as the parameter. For example:

```
$ ACS SET LIBRARY [JONES.HOTEL.ADALIB]
```

The current program library is the library to which compiler and ACS command operations apply. As such, the current program library is also the context for any units that are compiled or linked. For example, when the unit HOTEL is compiled, the compiler searches the current program library for the specification of the unit RESERVATIONS, because HOTEL mentions RESERVATIONS in a **with** clause.

The ACS SET LIBRARY command allows you to change the definition of the current program library from one library to another.

When working with several program libraries, you can determine which library is the current program library with the ACS SHOW LIBRARY command. For example:

```
$ ACS SHOW LIBRARY
%I, Current program library is USER:[JONES.HOTEL.ADALIB]
```

1.1.5 Compiling the Program

To compile VAX Ada compilation units, enter either the DCL ADA command or the ACS LOAD and COMPILE commands. The ADA and LOAD commands take one or more Ada source file names as parameters; the COMPILE command takes one or more Ada compilation unit names as parameters.

For example, the following ADA command compiles the files for the units RESERVATIONS and HOTEL. Because the ADA command assumes a .ADA file type by default, the file type is omitted.

```
$ ADA RESERVATIONS_, RESERVATIONS, HOTEL
```

Similarly, the following ACS LOAD and COMPILE commands compile the same set of files (again, .ADA is the default file type):

```
$ ACS LOAD RESERVATIONS*, HOTEL
$ ACS COMPILE HOTEL
```

Each time a compilation is successful, the program library is updated with information about the compilation units, as well as with files that are products of the compilation (object files and so on). One difference between the DCL ADA and ACS LOAD commands is that the ADA command fully compiles the units it processes; the LOAD command only partially compiles the units it processes. After entering an ACS LOAD command, you must subsequently enter an ACS COMPILE or RECOMPILE command to finish the processing.

For the ADA or LOAD command to execute successfully, you must have satisfied the following prerequisites:

- Defined a current default directory for your Ada source files (see Section 1.1.1)
- Created and set a current program library for the products of compilation (see Section 1.1.3)

For the ADA command to execute successfully, you must also have specified the files so that the units contained in the files are compiled in the correct order. The ACS LOAD command processes the units contained in the source files in any order, so it does not have this requirement.

You can determine the order of compilation by following the Ada rules for dependences among compilation units. For example, the order of compilation for the three compilation units of the hotel reservation program is as follows:

- The specification of RESERVATIONS must be compiled before the main procedure HOTEL because HOTEL names RESERVATIONS in a **with** clause.
- The specification of RESERVATIONS must be compiled before its body.
- The procedure HOTEL and the body of RESERVATIONS may be compiled in either order once the specification of RESERVATIONS has been compiled.

See Section 1.3.2 and the *VAX Ada Language Reference Manual* for more information on Ada order-of-compilation rules.

1.1.6 Displaying Unit Information

To display the contents of your program library, enter the ACS DIRECTORY command, specifying zero or more unit names as parameters. For example:

```
$ ACS DIRECTORY HOTEL, RESERVATIONS

HOTEL
    procedure body                  15-Apr-1989 14:53

RESERVATIONS
    package specification           15-Apr-1989 14:52
    package body                    15-Apr-1989 14:51

Total of 3 units.
```

The ACS DIRECTORY command identifies all compilation units that are part of the program library. Compilation units are listed alphabetically by unit name, and the date and time of the most recent compilation is given for each unit.

You can obtain information on how portable your program is by using the /PORTABILITY qualifier with the ACS SHOW PROGRAM command.

1.1.7 Linking the Program

Once you have compiled all of the units of a program into the current program library, you link the program by entering the ACS LINK command (not the DCL LINK command). You specify the unit name (not the file name) of the main program unit as the parameter. For example:

```
$ ACS LINK HOTEL
```

The ACS LINK command invokes the VAX Ada program library manager, which serves as the interface to the VMS Linker and performs the following link-related operations:

- Checks that a complete set of units exists for the unit specified (the main program), and that all of the units are current
- If the set of units is complete and current (see Sections 1.3.1.2 and 1.3.3), generates a temporary command file for the linker
- Invokes the linker

The linker uses the information in the command file to link the appropriate object modules and produces an executable image file (.EXE) with the same name as the main program. This image file is stored in your current default directory (not the current program library). Thus, in the example hotel reservation program, the resulting executable image file, HOTEL.EXE, is in the directory [JONES.HOTEL], not in the directory [JONES.HOTEL.ADALIB].

1.1.8 Executing the Program

To execute a successfully linked program, enter the DCL RUN command, specifying the name of the executable image file as the parameter. For example:

```
$ RUN HOTEL
```

Because the DCL RUN command assumes a .EXE file type by default, you can omit the file type of the executable image when you enter the DCL RUN command, as shown in this example.

1.1.9 Debugging the Program

If you expect to encounter run-time errors or need to check your Ada program as it is running, you can compile and link the program so that it will run under the control of the VMS Debugger when you execute the DCL RUN command. While you are executing your program under debugger control, you can set breakpoints, watchpoints, tracepoints, examine the contents of variables, control the operation of tasks, and so on (see Chapters 6 and 7).

The following commands show how the example hotel reservation program is compiled and linked for execution under debugger control. Because the /DEBUG qualifier is a default qualifier for the DCL ADA command, it is not shown here.

```
$ ADA HOTEL
$ ACS LINK/DEBUG HOTEL
$ RUN HOTEL

        VAX DEBUG Version 5.0-00

%DEBUG-I-INITIAL, language is ADA, module set to HOTEL
%DEBUG-I-NOTATAMAIN, type GO to get to start of main program
DBG>
```

Once you are in the debugger, you can obtain help on any of the debugger's features by typing the HELP command at the debugger prompt (DBG>). You can exit from the debugger at any time with the debugger EXIT command.

If you have compiled and linked a program with the /DEBUG qualifiers, and want to execute it without debugger control, you can enter the DCL RUN/NODEBUG command, as follows:

```
$ RUN/NODEBUG HOTEL
```

1.1.10 Compiling and Recompiling a Modified Program

If your program has been compiled once and then modified, you can compile it again by entering the DCL ADA command as described in Section 1.1.5. Alternatively, you can use the ACS COMPILE command. If the compilation order of the units in the program has not changed, enter the ACS COMPILE command, specifying the unit name of the main program. For example:

```
$ ACS COMPILE HOTEL
```

The ACS COMPILE command finds all of the compilation units that are required for the execution of the unit specified, automatically compiles any source files that have been modified, and recompiles any units that are made obsolete or incomplete by the compilation. (See Section 1.3.1.2 for more information on obsolete units, incomplete units, and recompilation.)

If you expect that the compilation order has changed, you can use the /PRELOAD qualifier with the ACS COMPILE command. You can add new units to an existing set of units by first compiling them into the library with the ADA command or loading them into the library with the ACS LOAD command, and then entering the ACS COMPILE/PRELOAD command.

If you have a set of units that have not been modified but are obsolete because a unit that they depend on has changed, you can recompile them using the ADA command, or you can use either the ACS RECOMPILE or COMPILE command. For example:

```
$ ACS RECOMPILE HOTEL
```

The COMPILE or RECOMPILE command finds all of the compilation units that are required for the execution of the unit specified, and recompiles any obsolete or incomplete units.

By entering the COMPILE and RECOMPILE commands with the /NODATE_CHECK qualifier, you can use them to force the compilation or recompilation of a set of units.

Like the DCL ADA command, the ACS COMPILE and RECOMPILE commands assume the /DEBUG qualifier by default.

See Chapter 3 for more information on using the ACS COMPILE and RECOMPILE commands.

1.2 Using the VAX Ada Program Library Manager

The VAX Ada program library manager is an interactive utility with DCL-like commands—ACS commands—that you enter to perform a variety of functions. The program library manager handles all of the program library operations associated with Ada compilation units and automates many of those functions for you. The program library manager also provides much of the user interface to the VAX Ada compiler and VMS Linker.

This section gives an overview of the ACS commands, and discusses the following topics:

- Entering ACS commands
- Exiting from the program library manager and interrupting ACS commands
- Defining synonyms for ACS commands
- Using DCL commands with program libraries

1.2.1 Overview of ACS Commands

ACS commands provide program library management, compilation, and linking operations. These operations are summarized in this section as follows:

- Table 1–1 summarizes program library management commands. (See Chapters 2 and 5 for more information on program library management.)
- Table 1–2 summarizes compilation and linking commands. (See Chapter 3 for more information on compilation; see Chapter 4 for more information on linking.)
- Table 1–3 summarizes additional ACS commands that are useful in the VMS environment.

Appendix A of this manual is a dictionary of all of the ACS commands. It provides details on the format, parameters, and qualifiers for each command. The same information is provided on line when you type ACS HELP at the DCL prompt ($).

NOTE

For completeness, the DCL ADA and DCL RUN commands are included in these tables. These are the only non-ACS commands presented.

Table 1–1: ACS Program Library Management Commands

Command	Function
CHECK	Forms the execution closure[1] of one or more compiled units, and checks the completeness and currency of the units in the closure.
CONVERT LIBRARY	Converts a VAX Ada Version 1.n program library to a VAX Ada Version 2.0 program library.
COPY FOREIGN	Copies a foreign (non-Ada) object file into the current program library as a library unit body.
COPY UNIT	Copies a compiled unit from one program library to the current program library.
CREATE LIBRARY	Creates a VAX Ada program library.
CREATE SUBLIBRARY	Creates a VAX Ada program sublibrary, which allows you to isolate the development of selected units.
DELETE LIBRARY	Deletes a program library and its contents.
DELETE SUBLIBRARY	Deletes a program sublibrary and its contents.
DELETE UNIT	Deletes one or more compiled units from the current program library.
DIRECTORY	Lists the units in the current program library. Displays information, such as the name and date-time of the last compilation, about one or more units in the current program library.
ENTER FOREIGN	Enters a reference (pointer) from the current program library to an external file as a foreign (non-Ada) library body.
ENTER UNIT	Enters a reference (pointer) from the current program library to a unit that has been compiled into another program library. "Entered" units can be used in the current program library as if they were actually in it.

[1]In simple terms, *execution closure* is the complete set of units that a given unit depends on, plus any other units needed for its execution. Currency and closure are discussed in Sections 1.3.1.2 and 1.3.3, respectively.

(continued on next page)

Table 1–1 (Cont.): ACS Program Library Management Commands

Command	Function
EXPORT	Creates an object file that contains the object code for one or more units in the current program library.
EXTRACT SOURCE	Obtains copies of source files contained in the current program library.
MERGE	Merges, into the parent library, new versions of one or more units from the sublibrary where they were modified. MERGE replaces the older, obsolete versions in the parent library.
REENTER	Enters current references to units that were compiled after they were last entered with the ENTER UNIT command.
REORGANIZE	Optimizes the organization of a program library.
SET LIBRARY	Defines a program library to be the current program library—that is, the library that is to be the compilation context, as well as the target library for compiler output and ACS commands in general.
SET PRAGMA	Redefines specified values of the library characteristics LONG_FLOAT, MEMORY_SIZE, and SYSTEM_NAME.
SHOW LIBRARY	Displays the name and characteristics of one or more program libraries.
SHOW PROGRAM	Displays information, such as dependence on other units, about the closure of one or more units in the current program library. Also displays a portability summary.
SHOW VERSION	Displays the version of the VAX Ada compiler and program library manager being used.
VERIFY	Performs a series of consistency checks on a program library to determine whether the library structure and library files are in valid form. Optionally corrects some of the inconsistencies detected.

Table 1–2: Compilation, Linking, and Execution Commands

Command	Function
	DCL Commands
ADA	Invokes the VAX Ada compiler to compile the specified Ada source files.
RUN	Executes the specified executable image file.
	ACS Commands
COMPILE	Forms the execution closure[1] of one or more specified units; checks the completeness and currency of the units in the closure; identifies units that have revised source files; compiles units that have revised source files; recompiles units that are obsolete or will be made obsolete. Completes incomplete generic instantiations.
LOAD	Loads (partially compiles) the units in the specifed Ada source files into the current program library as obsolete units; updates the current program library with unit dependence and source-file information.
LINK	Creates an executable image file for the specified units.
RECOMPILE	Forms the execution closure[1] of one or more specified units; checks the completeness and currency of the units in the closure; recompiles any obsolete units in the appropriate order to make them current. Completes incomplete generic instantiations.
SET SOURCE	Defines a source file search list for the COMPILE command.
SHOW SOURCE	Displays the source file search list used by the COMPILE command.

[1]In simple terms, *execution closure* is the complete set of units that a given unit depends on, plus any other units needed for its execution. *Currency* and *closure* are discussed in Sections 1.3.1.2 and 1.3.3, respectively.

Table 1–3: Additional ACS Commands

Command	Function
ATTACH	Switches control of your terminal from the current process running the VAX Ada program library manager (same as the DCL ATTACH command).
EXIT	Exits from the program library manager. You can also use CTRL/Z.
HELP	Invokes the VMS HELP facility to obtain information about ACS commands.
SPAWN	Creates a subprocess of the current process (same as the DCL SPAWN command).

1.2.2 Entering ACS Commands

You can enter ACS commands in two ways:

- By invoking the program library manager interactively
- In the form of one-line DCL commands

To use the program library manager interactively, you must first invoke it by typing ACS at the DCL prompt ($). The library manager responds with the ACS prompt. For example:

```
$ ACS
ACS>
```

Once you have invoked the program library manager, you can enter any ACS command. For example:

```
ACS> SET LIBRARY [JONES.HOTEL.ADALIB]
```

To enter an ACS command as a one-line DCL command, type the ACS prefix and then the ACS command line. For example:

```
$ ACS SET LIBRARY [JONES.HOTEL.ADALIB]
```

This form allows you to use DCL symbol substitution, parameter passing, and lexical functions in ACS commands (these DCL features are described in the *VMS DCL Concepts Manual* and *Guide to Using VMS Command Procedures*). For example:

```
$! CLG.COM -- DCL procedure for compile-link-go processing.
$! Parameter P1 is source file name and main program name.
$ ADA 'P1'
$ ACS LINK 'P1'
$ RUN 'P1'
```

Regardless of the ACS command format you choose, the program library manager prompts you for any required parameters that are missing.

If your ACS command is too long to fit on one line, you can continue the command by typing a hyphen (-) as the last character of a line. For example:

```
ACS> LINK/DEBUG/MAP/FULL/CROSS_REFERENCE -
_ACS> MY_MAIN_PROGRAM -
_ACS> DISK:[MATRIX.SHARE]MATHPACK.OLB/LIB
```

An ACS command can have a maximum of 1024 characters. Individual command lines can have a maximum of 256 characters.

1.2.3 Exiting from the Program Library Manager and Interrupting ACS Commands

If you are using the program library manager interactively, you can exit and return to DCL level by entering the ACS EXIT command or by pressing CTRL/Z at the ACS> prompt. For example:

```
ACS> EXIT
$
```

If you need to interrupt an ACS command before its execution has completed, press CTRL/C rather than CTRL/Y. CTRL/C interrupts the command in an orderly fashion, while CTRL/Y may not. In particular, use CTRL/C if the ACS command is one that alters the contents of a program library, for example, the ACS DELETE UNIT command. When you use CTRL/Y to interrupt an ACS command, control passes directly to DCL, and the program library may be left in an inconsistent state.

1.2.4 Defining Synonyms for ACS Commands

As with DCL commands, you can define synonyms (symbols) to abbreviate commonly used combinations of ACS commands and qualifiers. You can place these symbol definitions in your LOGIN.COM file so that they take effect whenever you log in to your system.

A synonym for an ACS command must have the prefix ACS$. Otherwise, the conventions are identical to those for defining synonyms for DCL commands (see the *VMS DCL Concepts Manual*). For example:

```
$ ACS$DB  == "DIRECTORY/BRIEF"
$ ACS$DF  == "DIRECTORY/FULL"
```

You can use these synonyms when working interactively with the program library manager. For example:

```
ACS> DB
HOTEL
QUEUE_MANAGER
RESERVATIONS
SCREEN_IO

Total of 7 units.
```

Note from this example that you use only the letters following the ACS$ prefix as the synonym.

1.2.5 Using DCL Commands with Program Libraries

Program libraries are implemented in VAX Ada as VMS directories. However, the file relationships inside a program library are quite different from those in a conventional VMS directory. Therefore, in general, you should use only ACS commands to manipulate program libraries and their contents.

You may need to use DCL commands in certain situations. For example, you may need to use the DCL SET PROTECTION command to change the protection of a library directory so that you can delete it (see Chapters 2 and 5). Similarly, you may need to use the DCL BACKUP command to copy or back up a program library (see Chapter 5).

1.3 Concepts and Terminology

The following sections summarize the basic concepts and terminology that apply to compilation and linking in the VAX Ada environment. These concepts are related to modular program development, which is a primary feature of the Ada language.

1.3.1 Program and Compilation Units

Program units are the functional building blocks of Ada programs. There are four kinds of program units: subprograms (procedures and functions), packages, tasks, and generic units. An Ada program generally consists of a *main program* and its related program units. A main program is always a subprogram.

To facilitate modular development, each program unit consists of a specification and sometimes a body. The specification contains only the declarations that need to be made visible to other program units; the body contains the implementation of the declarations in the specification.

The parts of Ada program units that can be compiled separately are called *compilation units*. Compilation units consist of the following:

- Package specifications and bodies
- Subprogram specifications and bodies
- Generic unit (subprogram and package) specifications and bodies
- Generic instantiations (subprogram and package) of generic units
- Subunits

NOTE

A task specification or body must be contained within a package or a subprogram before it can be compiled, except when the task body is a subunit.

The Ada language distinguishes between two classes of compilation units:

- *Library units* are the compilation units that are essential for program compilation. They consist of library unit specifications, or *library specifications* (consisting of subprogram, package, and generic specifications), generic instantiations, and subprogram bodies that do not have corresponding specifications.
- *Secondary units* are the compilation units that are not essential for program compilation, but they are essential for program linking and running. They consist of library unit bodies, or *library bodies* (consisting of subprogram, package, and generic bodies), and subunits.

Thus, Ada allows you to begin program development by designing and compiling a program consisting only of library units. Once you have a consistent program structure that you can compile, you can implement any corresponding secondary units (bodies and subunits), and then link and run the program. See Chapter 5 for more information.

1.3.1.1 Compilation Unit Dependences

During and after compilation, the compiler and program library manager maintain current data on the status of compilation units and the *dependences* among units. In this way, the compiler can enforce certain order-of-compilation rules (see Section 1.3.2), and the program library manager can manage the program library to support those rules.

Compilation unit dependences are derived from Ada's scope and visibility conventions:

- A library body depends on its library specification, if there is one.
- A subunit depends on its parent unit and therefore depends on its parent's associated library body and library specification.
- Each compilation unit depends on the library specifications of any units that are named in **with** clauses.

Compilation unit dependences can also be caused by the following:

- The value of the predefined constant SYSTEM.SYSTEM_NAME if the package SYSTEM is named in a **with** clause. (Chapter 5 describes this constant and its effects in more detail.)
- Calls of subprograms that have been specified with the pragma INLINE.
- Instantiations of generics that have been specified with the pragma INLINE_GENERIC.

1.3.1.2 Current and Obsolete Units

Whenever a unit is compiled, any dependent unit, as defined in Section 1.3.1.1, is made *obsolete* and must eventually be compiled again before it can be included in a set of units to be linked. For example, compiling a library specification makes the associated library body and any subunits obsolete; moreover, if the library specification is named in a **with** clause of a unit, that unit is also made obsolete, as are its dependent units. Incomplete instantiations (instantiations that were compiled before their corresponding generic body was compiled or recompiled) are also counted as obsolete units.

The program library manager keeps track of current and obsolete units. ACS commands such as SHOW PROGRAM and CHECK allow you to determine the status of the units in the current program library. If you try to link a set of units that contains any obsolete units, the program library manager warns you about those units and terminates the operation. Because obsolete units are a natural consequence of Ada's compilation rules (see Section 1.3.2), VAX Ada provides the ACS COMPILE and RECOMPILE

commands. These commands automatically find the set of units that need to be compiled to make an obsolete unit current, and then compile that set in the right order. This process makes the units *current*.

NOTE

The verb *to recompile* is used in a restricted sense in this manual; it means to make a set of obsolete units current.

1.3.1.3 Unit and File-Name Conventions

While developing programs in the VAX Ada environment, you need to recognize the distinction between source files and units. A source file (having a default file type of .ADA) can contain several compilation units. However, after a file is compiled, the program library manager maintains information about the individual units, and most of the ACS commands operate on units (not on source files).

If you have one source file for all of your compilation units, the name of the file will be different from most, if not all, of the units. Because most program library manager commands accept unit names and give information about units, having one source file with a different name from most units can become confusing. To keep the distinction between source files and compilation units clear, use a separate source file for each compilation unit.

Use of a separate source file for each compilation unit also promotes efficient use of the compiler. For example, every time a unit is compiled, any dependent unit in the program library is made obsolete and must be recompiled. Thus, if you have two library specifications in the same source file, every time you modify one specification, you must compile both in the same compilation. Then, the units that depend on both specifications become obsolete and must be recompiled. If the specifications were in separate source files, only the modified specification would be compiled, and only the units that depend on the modified specification would become obsolete and have to be recompiled.

When you use separate source files for individual compilation units, you should follow file-name conventions that parallel the Ada language rules for naming compilation units. For example, although a library specification and its library body are distinct compilation units, they share the same name, called the *unit name*. All of the unit names in a program library must be unique. Similarly, all of the subunit names associated with a given ancestor unit must be unique. (Every subunit mentions the name of its *parent unit*, and the top-level parent in a hierarchy of subunits is the *ancestor unit*.)

To support these rules, the following file-name conventions are recommended. These conventions are consistent with program library manager and VMS file-name conventions.

- The name of the source file for a *library specification* should be the name of the unit, followed by a trailing underscore character (_): for example, SCREEN_IO_.ADA.
- The name of the source file for a *library body* should be the name of the unit (without a trailing underscore): for example, SCREEN_IO.ADA.
- The name of the source file for a *library generic instantiation* should be the name of the instantiation: for example, INTEGER_TEXT_IO.ADA.
- The name of the source file for a *subunit* should be the name of the ancestor unit, followed by two underscore characters, followed by the name of the subunit: for example, SCREEN_IO__INPUT.ADA (where INPUT is a subunit of SCREEN_IO).

Table 1–4 shows the conventions for naming source files by comparing unit names with source file names. The names in the table represent the following arbitrary set of units:

- Package specification and body SCREEN_IO
- Generic package declaration and body MATH
- Generic instantiation HOTEL_MATH
- Subunit INPUT (of SCREEN_IO)
- Subunit BUFFER (of INPUT)

Table 1–4: Conventions for Naming VAX Ada Source Files

Compilation Unit	Ada Unit Name	Ada Source File Name
package SCREEN_IO		
specification	SCREEN_IO	SCREEN_IO_
body	SCREEN_IO	SCREEN_IO
generic package MATH		
declaration	MATH	MATH_
body	MATH	MATH
generic instantiation	HOTEL_MATH	HOTEL_MATH

(continued on next page)

Table 1–4 (Cont.): Conventions for Naming VAX Ada Source Files

Compilation Unit	Ada Unit Name	Ada Source File Name
subunits		
INPUT	SCREEN_IO.INPUT	SCREEN_IO__INPUT
BUFFER	SCREEN_IO.INPUT.BUFFER	SCREEN_IO__BUFFER

1.3.2 Order-of-Compilation Rules

The VAX Ada compiler and program library manager enforce the rules governing the order in which compilation units are compiled. These order-of-compilation rules stem from Ada's scope and visibility conventions, which create the dependences among units described in Section 1.3.1.1. The rules are as follows:

- You can compile a given unit only *after* compiling all of the library specifications named in that unit's context clause.
- You can compile a library body only *after* compiling its library specification. However, the body of a nongeneric library subprogram can also serve as its own library specification, and therefore does not necessarily depend on a separately compiled specification.
- You can compile a subunit only *after* compiling its parent unit.

In summary, a unit must be compiled *before* any of its dependent units.

If you follow these rules, then the following additional rules are true:

- You can submit the compilation units of a program to the compiler in one or more compilations (invocations of the compiler). Also, you can submit one or more compilation units of a program at any one time. The units of any one compilation are compiled in the given order, whether submitted in one or more files. Thus, a pragma that applies to the whole of a compilation must appear before the first unit of that compilation.
- Units can be compiled in an otherwise arbitrary order relative to each other. For example, compiling a subunit affects only its subunits, if any; compiling a library body generally does not affect any other units except its own subunits, if any. However, compiling a library body does affect other units in the following three cases:
  - If a pragma INLINE or equivalent /OPTIMIZE qualifier option applies to a subprogram, then compiling the library body containing the subprogram body makes obsolete any unit in which a call of the subprogram is expanded inline.

- If a pragma INLINE_GENERIC or equivalent /OPTIMIZE qualifier option applies to a generic unit or to an instance of a generic unit, then compiling the generic body makes obsolete any unit in which an instantiation of the generic is expanded inline.
- If an inline pragma or equivalent /OPTIMIZE qualifier option does not apply, then compiling a generic body makes all instantiations of the generic incomplete. However, units that contain instantiations of the generic do not become obsolete. (See Chapter 3 for more information on completing incomplete generic instantiations.)

If you follow these rules when you compile a unit or set of units, and no other errors are detected during the compilation, then the program library is *updated* with information on all of the units in the compilation. If the compilation is unsuccessful for any reason, no updating is done.

Although the VAX Ada compiler always processes compilation units in a manner that is consistent with Ada's order-of-compilation rules, observance of the compilation rules does not ensure that the set of units in a program library is *current*. Nor does observance of the rules ensure that the set of units is *complete*. For example, a library body or a subunit may still be missing from the program library, or may have been made obsolete by a previous compilation. If you try to link an incomplete set of units, the program library manager warns you about the missing units, and terminates the operation.

Obsolete units are discussed in Section 1.3.1.2; what constitutes a complete set of units is discussed in Section 1.3.3.

1.3.3 Closure

When you compile a given unit, the compiler identifies any unit that the given unit depends on, as specified in Section 1.3.1.1, and determines whether that unit is defined and current in the current program library. For example, if the given unit is a library body, the compiler looks for the unit's specification.

Any unit that a given unit depends on may itself depend on another unit, which must also be defined in the current program library. The total set of library units that the given unit depends on, directly and indirectly, is called the *compilation closure* of that unit. Thus, the compilation closure of a given unit consists of all the units that must be defined and current in the current program library before you can compile that unit.

To link a program into an executable image, the *execution closure* of the main program must be formed. The execution closure consists of the compilation closure plus all associated secondary units (library bodies and subunits). A set of units is *complete* when no units in the execution closure are missing.

A number of ACS commands operate on the execution closure of a specified set of units—for example, the ACS CHECK, COMPILE, COPY UNIT/CLOSURE, ENTER UNIT/CLOSURE, EXPORT, LINK, RECOMPILE, and SHOW PROGRAM commands.

NOTE

In this manual, the term *closure* is used to signify execution closure, unless otherwise specified.

The execution closure of a specified set of compilation units is defined formally as the smallest set of units with the following properties:

- All the specified units are contained in the closure.
- For any given unit in the closure, the following are also contained in the closure, as applicable:
  - Its specification, if the given unit is a body
  - Its body, if the given unit is a specification
  - Its immediate subunits, if any
  - Its immediate parent unit, if the given unit is a subunit
  - All units named by the given unit in its **with** clause

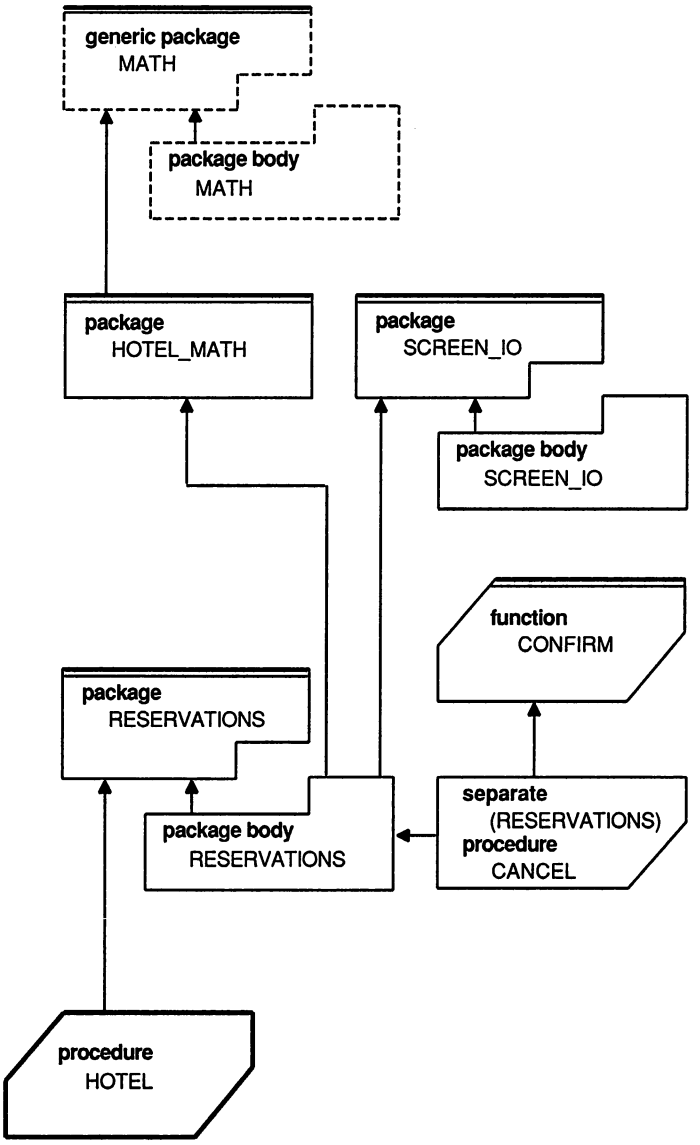
A unit that names a given unit in its **with** clause is not part of the execution closure of the given unit.

Figure 1–4 shows one possible configuration of an extended version of the HOTEL reservation program. The units involved are the library packages RESERVATIONS, SCREEN_IO, and HOTEL_MATH, the library subprograms HOTEL and CONFIRM, and the subunit RESERVATIONS.CANCEL. Arrows point from dependent units to the units they depend on.

The units shown in Figure 1–4 form the following closures:

- The closure of the unit CONFIRM consists of the function CONFIRM.
- The closure of the specification or body of the package SCREEN_IO consists of the specification and body of the package SCREEN_IO.

Figure 1–4: Sample Compilation Units Used to Show Closure



- The closure of the specification, body, or subunit of the package RESERVATIONS consists of all of the units shown, except for the procedure HOTEL.
- The closure of the procedure HOTEL consists of all of the units shown.

The following command recompiles any of the units shown that are obsolete, except HOTEL (the closure of RESERVATIONS does not include HOTEL):

```
$ ACS RECOMPILE RESERVATIONS
```

The following command recompiles any of the units shown that are obsolete (the closure of HOTEL includes all of the units):

```
$ ACS RECOMPILE HOTEL
```

Chapter 2

Working with VAX Ada Program Libraries and Sublibraries

Ada compilations are performed in the context of a program library. The program library manager and compiler use the program library to maintain information about compilation units.

A VAX Ada *program library* is a dedicated VMS directory that contains a set of files for each compilation unit successfully compiled. A VAX Ada *program sublibrary* is a program library that has a parent library. Units in a sublibrary are compiled in the context of both the sublibrary and the parent library, but only the sublibrary is updated.

NOTE

Because program libraries and sublibraries are so similar, many library and compilation unit operations have the same effect. Thus, this chapter uses the term *library* to denote a sublibrary as well as a program library. The terms *program library* and *sublibrary* are used only where emphasis is needed or a distinction must be made.

When your library context is a sublibrary, the units in the sublibrary and parent libraries are visible in a fashion analogous to multiple panes of glass. The units in the sublibrary appear on the top pane, units in the immediate parent library appear on the next pane, units in the parent of the immediate parent appear on a following pane, and so on. Then, units by the same name hide each other such that a unit in a parent library is hidden (made not visible) by a unit of the same name in a closer sublibrary. Thus, the search for a unit begins with the closest pane of glass (the sublibrary) and follows through the parent panes until the unit is found.

You can organize program libraries and sublibraries to suit the needs of your project. For example, you can store the compilation units of an entire Ada program in a single program library, or you can distribute them among a number of program libraries. Sublibraries are designed to allow you to isolate particular compilation units so that you can develop them individually.

This chapter explains how you can work with program libraries and sublibraries using ACS commands. Chapter 5 covers additional topics related to program library management and maintenance.

NOTE

The information in this chapter is task oriented. For full details on the format, parameters, and qualifiers of the various ACS commands, see Appendix A. For information on the implementation of VAX Ada program libraries and sublibraries, see Appendix D.

2.1 Program Library and Sublibrary Operations

The following sections describe a number of program library and sublibrary operations:

- Creating a program library or sublibrary
- Defining the current program library
- Identifying the current program library
- Obtaining library information
- Controlling library access
- Deleting a program library or sublibrary

In general, the effect of these operations on program libraries and sublibraries is the same. When the effect is different, information is provided, as appropriate. See Section 2.3.1 for a summary of the commands where the effect is different.

See Chapter 5 for information on how to configure, protect, and maintain program libraries and sublibraries.

NOTE

Use only ACS commands (not DCL commands) to manipulate program libraries and sublibraries. Exceptions to this rule are noted where appropriate.

2.1.1 Creating a Program Library or Sublibrary

To create a program library, enter the ACS CREATE LIBRARY command, specifying a VMS directory as a parameter. For example:

```
$ ACS CREATE LIBRARY [JONES.HOTEL.ADALIB]
```

To create a sublibrary, enter the ACS CREATE SUBLIBRARY command, specifying a VMS directory as a parameter, and optionally specifying the parent library with the /PARENT qualifier. For example:

```
$ ACS CREATE SUBLIBRARY/PARENT=[JONES.HOTEL.ADALIB] -
_$ [JONES.HOTEL.SUBLIB]
```

When creating a sublibrary, you can specify any previously created program library or sublibrary to be the parent library. If you omit the /PARENT qualifier, the current program library is defined to be the parent library by default. See Section 2.1.2 for information on defining and identifying the current program library; see Section 2.1.4 for information on identifying the parent of a sublibrary.

NOTE

By using concealed-device logical names and rooted directory syntax for program library and sublibrary directories, you can make the maintenance of program libraries and sublibraries easier. In particular, you can change the parent of a sublibrary. See Section 2.3.3 and Chapter 5 for more information.

The ACS CREATE LIBRARY and CREATE SUBLIBRARY commands are the *same* in the following respects:

- They create the specified VMS directory (if it does not already exist).
- They cannot be used across DECnet unless the VMS directory for the library already exists.
- They create the library, but do not automatically make it a target for compilation and ACS commands. To use the library, you must enter the ACS SET LIBRARY command (see Section 2.1.2).
- They cause the library directory to inherit the default system file protection. Both commands have a /PROTECTION qualifier, which allows you to change the default. See Chapter 5 for more information on library protection.

The ACS CREATE LIBRARY and CREATE SUBLIBRARY commands are *different* in the following respects:

- The CREATE LIBRARY command initializes the program library to be self-contained. The CREATE SUBLIBRARY command puts a reference to the parent library in the sublibrary.
- The CREATE LIBRARY command enters the Ada predefined units into the newly created program library by default. The CREATE SUBLIBRARY command does not enter the Ada predefined units into the newly created sublibrary.
- When you create a program library, the following system characteristics are in effect by default:
  - LONG_FLOAT = G_FLOAT
  - MEMORY_SIZE = 2147483647
  - SYSTEM_NAME = VAX_VMS

  When you create a sublibrary, the sublibrary inherits the defaults of its parent library or sublibrary. The CREATE LIBRARY and CREATE SUBLIBRARY commands have qualifiers that allow you to override these defaults. See Chapter 5 and the descriptions of these commands in Appendix A for more information; see also the description of the ACS SET PRAGMA command, which allows you to change the system characteristics for existing libraries or sublibraries.

A program library or sublibrary is meant to hold only the files needed for the program library manager. You should not use it for any other purpose. For example, you should keep it distinct from any working directory (such as the current default directory) where you store and edit your source files.

2.1.2 Defining the Current Program Library

The current program library is the target library for compilation and many ACS commands. To define a library as the *current program library*, enter the ACS SET LIBRARY command, specifying the VMS directory specification for the library as the parameter. For example:

```
$ ACS SET LIBRARY [JONES.HOTEL.ADALIB]
```

The program library manager assigns the directory specification provided in the SET LIBRARY command to the process logical name ADA$LIB. Both the program library manager and the compiler use that logical name to maintain the current program library context when performing various operations.

Note that if you specify an invalid library directory specification, the program library manager issues a diagnostic message and then sets the library (and assigns ADA$LIB) to the invalid specification. This behavior is designed to protect you from incorrectly modifying the wrong library with subsequent ACS commands.

2.1.3 Identifying the Current Program Library

To identify the current program library, enter the ACS SHOW LIBRARY command without a parameter. For example:

```
$ ACS SHOW LIBRARY
%I, Current program library is USER:[JONES.HOTEL.ADALIB]
```

2.1.4 Obtaining Library Information

To obtain information about the current program library, enter the ACS SHOW LIBRARY command with one of a number of qualifiers. For example, you can use the /FULL qualifier to determine a library's system characteristics:

```
$ ACS SHOW LIBRARY/FULL
%I, Current program library is USER:[JONES.HOTEL.ADALIB]

Program library USER:[JONES.HOTEL.ADALIB]

Created:          13-Apr-1989 15:51, by VAX Ada 2.0
Last reorganized: 16-Apr-1989 13:46

Pragmas that affect STANDARD and SYSTEM:

    pragma LONG_FLOAT(G_FLOAT)
    pragma MEMORY_SIZE(2147483647)
    pragma SYSTEM_NAME(VAX_VMS)
```

You can also use the /FULL qualifier to determine the parent of a sublibrary.

To obtain information about libraries that are not the current program library, enter the ACS SHOW LIBRARY command, specifying the libraries of interest as parameters. For example:

```
$ ACS SHOW LIBRARY/FULL [JONES.HOTEL.SUBLIB.SUBSUBLIB]

Program library USER:[JONES.HOTEL.SUBLIB.SUBSUBLIB]

    Sublibrary
        of USER:[JONES.HOTEL.SUBLIB]
        of USER:[JONES.HOTEL.ADALIB]

Created:          13-Apr-1989 15:53, by VAX Ada 2.0
Last reorganized: <No reorganization date>
```

```
Pragmas that affect STANDARD and SYSTEM:

    pragma LONG_FLOAT(G_FLOAT)
    pragma MEMORY_SIZE(2147483647)
    pragma SYSTEM_NAME(VAX_VMS)
```

To display the contents of a library, you can use the /UNITS qualifier on the ACS SHOW LIBRARY command. To display the contents of the current program library, you can use the ACS DIRECTORY command. The results of the SHOW LIBRARY/UNITS command and the DIRECTORY command are the same. However, you can apply the DIRECTORY command only to the current program library; you can apply the SHOW LIBRARY/UNITS command to any library. See Section 2.2.2 for more information on the ACS DIRECTORY command.

2.1.5 Controlling Library Access

The ACS SET LIBRARY command has two qualifiers that allow you to temporarily control library access:

- The /READ_ONLY qualifier temporarily allows you to access libraries in a read-only manner
- The /EXCLUSIVE qualifier temporarily limits library access to one process

To use either qualifier, execute the ACS SET LIBRARY command interactively from the program library manager. For example:

```
ACS> SET LIBRARY/READ_ONLY DISK:[SMITH.SHARE.ADALIB]

ACS> SET LIBRARY/EXCLUSIVE [JONES.HOTEL.ADALIB]
```

When you use these qualifiers, they remain in effect until you exit from the program library manager or until you execute another ACS SET LIBRARY command.

The following sections describe the use of these qualifiers in more detail. See Chapter 5 for information on permanently controlling library access using file and directory protection mechanisms.

2.1.5.1 Read-Only Access

The /READ_ONLY qualifier to the ACS SET LIBRARY command is useful when you want to limit your access to a library for reading only. For example, the /READ_ONLY qualifier is useful when you want to protect yourself from accidentally modifying a library to which you also have write access. (Read access is also determined by the protection set for the library directory; see Section 2.1.1 and Chapter 5.)

The /READ_ONLY qualifier has an effect only when you enter the ACS SET LIBRARY command interactively. After executing the ACS SET LIBRARY command with the /READ_ONLY qualifier, you have read-only access to that library until you exit from the program library manager or until you enter another SET LIBRARY command. Read-only access means that you can enter only those ACS commands that do not require write access. For example:

- CHECK
- DIRECTORY
- EXPORT
- EXTRACT SOURCE
- LINK
- SHOW LIBRARY
- SHOW PROGRAM
- SHOW VERSION
- VERIFY

In the following example, the /READ_ONLY qualifier limits the user to read-only access of a general project library:

```
ACS>  SET LIBRARY/READ_ONLY [PROJ.ADALIB]
%I, Current program library is DISK:[PROJ.ADALIB]
ACS>  CHECK HOTEL
%I, All units current, no recompilations required
ACS>  EXPORT HOTEL
    .
    .
    .
ACS>  EXIT
```

2.1.5.2 Exclusive Access

When more than one process has both read and write access to a library, although the library will not be corrupted, there is some risk that it may be updated in a way that gives unexpected results. For example, a unit can become obsolete the moment it enters the library because a unit that it depends on has been simultaneously updated. You can use the ACS SET LIBRARY/EXCLUSIVE command to make sure that your process is the only one updating a library at a particular time.

For example, on a multiperson project you can use this command to temporarily protect the project program library while you enter, copy, or link units from another library:

```
$ ACS
ACS> CREATE LIBRARY [HOTEL.TEST]
%I, Library DISK:[HOTEL.TEST] created
ACS> SET LIBRARY/EXCLUSIVE [HOTEL.TEST]
%I, Current program library is DISK:[HOTEL.TEST]
 . . .  Enter, copy, or link units  . . .
ACS> EXIT
```

The /EXCLUSIVE qualifier is also useful when you are repairing (ACS VERIFY/REPAIR) or reorganizing (ACS REORGANIZE) a library.

After executing an ACS SET LIBRARY command with the /EXCLUSIVE qualifier, you have exclusive read and write access to that library until you exit from the program library manager or until you enter another SET LIBRARY command. If your process has exclusive access to a library, no other process can access that library for either reading or writing.

Note that while the /EXCLUSIVE qualifier is in effect, batch jobs (subprocess or your own) will not be able to access the library. In other words, this qualifier will affect the behavior of any commands (ACS LOAD, COMPILE, RECOMPILE, and so on) that process in batch mode by default.

You cannot execute the ACS SET LIBRARY command with the /EXCLUSIVE qualifier while another process is accessing the specified library. You also cannot use the /EXCLUSIVE qualifier across DECnet.

2.1.6 Deleting a Program Library or Sublibrary

To delete a program library or sublibrary, enter the ACS DELETE LIBRARY or DELETE SUBLIBRARY command, specifying a VMS directory as a parameter. The directory you specify must be a VAX Ada library that was previously created with the ACS CREATE LIBRARY or CREATE SUBLIBRARY command.

For example:

```
$ ACS DELETE LIBRARY [JONES.TEMP.ADALIB]
```

You cannot use the ACS DELETE LIBRARY command to delete a sublibrary; similarly, you cannot use the ACS DELETE SUBLIBRARY command to delete a program library.

NOTE

Use the ACS DELETE LIBRARY and DELETE SUBLIBRARY commands with caution when you have program sublibraries. A parent library does not contain references to its sublibraries; therefore, when you delete a program library or sublibrary, you will not be warned of the existence of any sublibraries.

The effect of either command is to delete the contents of the library. If there are no more files in the library directory, and if the directory is not delete protected against the owner, then the directory is also deleted (by default, the VMS operating system protects a directory against deletion by its owner). If the directory still contains other files, or if the directory is delete protected against the owner, then the directory is not deleted. You must use the DCL DELETE command to first empty and then delete the directory. If the library directory is delete protected against the owner, you must use the DCL SET PROTECTION command to change the protection before you can delete the directory.

The ACS CREATE LIBRARY and CREATE SUBLIBRARY commands cause a library directory to inherit the default system file protection. Both commands have a /PROTECTION qualifier that allows you to specify the protection when you create the library or sublibrary (see Chapter 5 for more information on library protection).

2.2 Unit Operations

The following sections describe a number of unit operations:

- Obtaining information about the units in a library
- Checking units for currency and completeness
- Sharing units among different libraries
- Putting non-Ada "units" into a library
- Deleting units

In general, the effect of these operations on program libraries and sublibraries is the same. When the effect is different, information is provided, as appropriate. For a summary of the commands where the effect is different, see Section 2.3.1.

NOTE

Use only ACS commands (not DCL commands) to manipulate units in program libraries and sublibraries.

2.2.1 Specifying Units in ACS Commands

ACS commands that operate on compilation units accept one or more *unit names*, not *file names*, as parameters. When you enter ACS commands involving compilation units, observe the following conventions:

- You can specify a single unit name, or a list of unit names separated by commas (,). For example:

```
$ ACS DIRECTORY SCREEN_IO, RESERVATIONS.CANCEL
```

- You can use the standard VMS wildcard characters in many ACS commands. The wildcarding rules are similar to those for VMS file specifications (see the *VMS DCL Dictionary*). The percent sign (%) matches any single character in the position that the percent sign occupies in the unit name. The asterisk (*) matches zero or more characters in the position that the asterisk occupies in the unit name. Wildcard matching treats the unit name as a string. In a unit name, the period character (.) has no special standing as a punctuation character. For example, the following command displays information about the unit RESERVATIONS and any of its subunits:

```
$ ACS DIRECTORY RESERVATIONS*
```

By default, ACS commands usually operate on groups of related units, such as the specification and the body (for example, ACS DIRECTORY or DELETE UNIT) or the execution closure of the specified units (for example, ACS CHECK). The exact behavior reflects the typical use of the command.

Qualifiers are available to modify the default behavior. For example, the ACS DELETE UNIT/BODY_ONLY command deletes the body without affecting the specification; the ACS COPY UNIT/CLOSURE command copies the closure of the specified units.

Commands that operate on several units provide /LOG and /CONFIRM qualifiers. The /LOG qualifier allows you to control whether or not information about an operation is displayed when the operation is performed. The /CONFIRM qualifier allows you to confirm that an operation should be carried out for one or more units involved in the operation. For example, the ACS MERGE/LOG command displays a list of the units being merged. The ACS DELETE UNIT/CONFIRM command asks you for confirmation before deleting each of the units specified in the command.

2.2.2 Displaying General Unit Information

You enter the ACS DIRECTORY command to list units in the current program library and display general information about them. The ACS DIRECTORY command lists compilation units alphabetically by unit name. Subunit names are expressed using selected component notation. For example:

```
$ ACS DIRECTORY *QUEUE, HOTEL, SCREEN_IO*
GUEST_QUEUE
    package instantiation           15-Apr-1989 15:25   <entered>

QUEUE
    generic package                 15-Apr-1989 15:25   <entered>
    generic package body            15-Apr-1989 15:25   <entered>

HOTEL
    procedure body                  15-Apr-1989 15:26

SCREEN_IO
    package specification           15-Apr-1989 15:25
    package body                    15-Apr-1989 15:25

SCREEN_IO.INPUT
    procedure body                  15-Apr-1989 15:25

SCREEN_IO.INPUT.BUFFER
    function body                   15-Apr-1989 15:25

SCREEN_IO.OUTPUT
    procedure body                  15-Apr-1989 15:25

Total of 9 units.
```

As shown in this example, the ACS DIRECTORY command identifies units by name and by kind (package specification, procedure body, and so on). The display also shows the date and time of the compilation of each unit, and identifies entered units.

By using an asterisk wildcard (*) or by omitting its parameter, you can use the ACS DIRECTORY command to list all of the units that are defined in the current program library.

By using qualifiers (/BRIEF, /FULL, and /ENTERED), you can control the level of information displayed.

If the current program library is a sublibrary, the ACS DIRECTORY command shows only the units in the sublibrary; it does not show units in any of the parent libraries.

2.2.3 Displaying Dependence and Portability Information

The ACS SHOW PROGRAM command displays information about the execution closure of a set of compilation units in the current program library. In particular, the ACS SHOW PROGRAM command displays information about unit dependences (the use of **with** clauses), potential portability constraints, unit currency, and so on.

Because it displays information about the execution closure of a set of units, the ACS SHOW PROGRAM command displays information about all of the relevant units, even if the current program library is a sublibrary and some of the units are in parent libraries.

The ACS SHOW PROGRAM command lists compilation units alphabetically by unit name. Subunit names are expressed using selected component notation. The command has qualifiers (/BRIEF, /FULL, and /PORTABILITY) that allow you to specify the level of information and the kind of information to be displayed.

You can use the /BRIEF qualifier with the ACS SHOW PROGRAM command to limit the display to the following information:

- The name of the program and the time the ACS SHOW PROGRAM command was executed
- The name of the library
- The values of pragmas that affect the predefined packages STANDARD and SYSTEM
- The name and kind of each unit contained in the closure
- The compilation date for each unit in the closure
- The directory containing the source file for each unit or the library from which the unit was entered

For example:

```
$ ACS SHOW PROGRAM/BRIEF SCREEN_IO

SCREEN_IO
15-Apr-1988 15:26

Program library USER:[JONES.HOTEL.ADALIB]

Created:          15-Apr-1989 14:47, by VAX Ada 2.0
Last reorganized: 15-Apr-1989 15:35

Pragmas that affect STANDARD and SYSTEM:

   pragma LONG_FLOAT(G_FLOAT)
   pragma MEMORY_SIZE(2147483647)
   pragma SYSTEM_NAME(VAX_VMS)

The closure of the specified units is:

IO_EXCEPTIONS
  Package specification
    Compiled:     13-Apr-1989 23:35
    Entered from: SYS$COMMON:[SYSLIB.ADALIB]

SCREEN_IO
  Package specification
    Compiled:   15-Apr-1989 15:25
    Source file: 1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO_.ADA;1

  Package body
    Compiled:   15-Apr-1989 15:25
    Source file: 1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO.ADA;1

SCREEN_IO.INPUT
  Procedure body
    Compiled:   15-Apr-1989 15:25
    Source file: 1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO__INPUT.ADA;1

SCREEN_IO.INPUT.BUFFER
  Function body
    Compiled:   15-Apr-1989 15:25
    Source file: 1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO__BUFFER.ADA;1

SCREEN_IO.OUTPUT
  Procedure body
    Compiled:   15-Apr-1989 15:25
    Source file: 1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO__OUTPUT.ADA;1

SYSTEM
  Builtin package

TEXT_IO
  Package specification
    Compiled:     13-Apr-1989 23:37
    Entered from: SYS$COMMON:[SYSLIB.ADALIB]
```

```
  Package body
    Compiled:       13-Apr-1989 23:37
    Entered from:   SYS$COMMON:[SYSLIB.ADALIB]
```

You enter the ACS SHOW PROGRAM command with no qualifiers to add dependence information (**with** list information) to the display. For example:

```
$  ACS SHOW PROGRAM SCREEN_IO

    .
    .
    .
SCREEN_IO
   Package specification
      Compiled:       15-Apr-1989 15:25
      Source file:     1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO_.ADA;1

   Package body
      Compiled:       15-Apr-1989 15:25
      Source file:     1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO.ADA;1
      With list:      TEXT_IO
   .
   .
   .
```

You can use the /PORTABILITY qualifier to display a portability summary (see Chapter 5 for details on the kinds of information that appear in the portability summary). For example:

```
$  ACS SHOW PROGRAM/PORTABILITY SCREEN_IO
 . . .
SCREEN_IO
   Package specification
      Compiled:     15-Apr-1989 15:25
      Source file:   1-Sep-1988 10:39   USER:[PROJ]SCREEN_IO_.ADA;1

   Package body
      Compiled:     15-Apr-1989 15:25
      Source file:   1-Sep-1988 10:39   USER:[PROJ]SCREEN_IO.ADA;1
      With list:      TEXT_IO
   .
   .
   .
PORTABILITY SUMMARY

predefined SHORT_INTEGER or SHORT_SHORT_INTEGER
                          SYSTEM spec

with SYSTEM               TEXT_IO body

predefined F_FLOAT, D_FLOAT, G_FLOAT or H_FLOAT*
                          TEXT_IO body

enumeration representation clause
                          SYSTEM spec
                          TEXT_IO spec
```

```
length SIZE representation clause
                        SYSTEM spec

record representation clause
                        SYSTEM spec

pragma PACK             SYSTEM spec

pragma IMPORT_EXCEPTION*
                        IO_EXCEPTIONS spec

pragma IMPORT_FUNCTION* TEXT_IO spec

pragma IMPORT_PROCEDURE*
                        TEXT_IO

pragma INTERFACE        TEXT_IO

        where * indicates an implementation-defined feature
```

2.2.4 Checking Unit Currency and Completeness

The VAX Ada compiler processes compilation units in a manner that is consistent with Ada's rules. However, observance of the compilation rules does not ensure that the execution closure of a set of units in a program library is either complete or current (see Chapter 1 for definitions of closure, completeness, and currency). For example, a library package body may still be missing from the program library, or a library specification may have been modified and compiled more recently than some dependent units, making the dependent units obsolete and in need of recompilation.

If you try to link a program that has missing or obsolete units, these errors will be automatically detected, and the operation will be terminated. Alternatively, you can enter the ACS CHECK command to check the completeness and currency of the units in your program before you link it.

The ACS CHECK command accepts one or more unit names as parameters, and then searches the execution closure of the set of units specified for missing or obsolete units. Because it searches for the execution closure of a set of units, the ACS CHECK command searches the current program library and any parent libraries, if the current program library is a sublibrary. Note, however, that for units specified with wildcards, the ACS CHECK command searches only the current program library for the specified units.

If the set of units in the closure is both complete and current, the following message is displayed:

```
%I, All units current, no recompilations required
```

If the ACS CHECK command finds that a unit, such as a subunit, is missing, a message like the following is displayed:

```
%E, Separate procedure body SCREEN_IO.OUTPUT not found in library
```

For example, consider the following situation:

- The body of RESERVATIONS names SCREEN_IO in a **with** clause.
- The specification of SCREEN_IO has been compiled more recently than the specification, body, and subunits of RESERVATIONS.

The following ACS CHECK command identifies the obsolete units that need to be recompiled. Note that because SCREEN_IO is in the execution closure of RESERVATIONS, the CHECK command also identifies the missing subunit SCREEN_IO.OUTPUT.

```
$ ACS CHECK RESERVATIONS
%E, Separate procedure body SCREEN_IO.OUTPUT not found in
        library
%E, Obsolete library units are detected

%I, The following units need to be recompiled:
RESERVATIONS
    package specification           15-Apr-1989 15:44
    package body                    15-Apr-1989 15:44
SCREEN_IO
    package body                    15-Apr-1989 15:44
SCREEN_IO.INPUT
    procedure body                  15-Apr-1989 15:44
SCREEN_IO.INPUT.BUFFER
    function body                   15-Apr-1989 15:44
RESERVATIONS.RESERVE
    procedure body                  15-Apr-1989 15:44
RESERVATIONS.RESERVE.BILL
    procedure body                  15-Apr-1989 15:44
RESERVATIONS.CANCEL
    procedure body                  15-Apr-1989 15:44

%I, The following units have missing subunits:
SCREEN_IO
    package body                    15-Apr-1989 15:44
```

You can also use the ACS CHECK command to identify units that depend on generic bodies. A unit that depends on a generic body must be completed with the ACS RECOMPILE or COMPILE command under the following conditions:

- After the generic body is first compiled
- Whenever the generic body is compiled again

For example, consider the following situation:

- The package GUEST_QUEUE is a library instantiation of the generic package QUEUE.
- The specification of the package QUEUE_MANAGER names GUEST_QUEUE in a **with** clause.

If the generic body of package QUEUE is compiled more recently than its instantiation GUEST_QUEUE, then GUEST_QUEUE becomes incomplete and must be recompiled:

```
$ ADA QUEUE_, QUEUE, GUEST_QUEUE, QUEUE_MANAGER_, QUEUE_MANAGER
    .
    .
    .
$ ADA QUEUE
$ ACS CHECK QUEUE_MANAGER

%E, Obsolete library units are detected

%I, The following units need to be completed (use ACS COMPILE or
    ACS RECOMPILE):
GUEST_QUEUE
    package instantiation           15-Apr-1989 15:43
```

Note that when GUEST_QUEUE is completed, QUEUE_MANAGER, the unit that depends on GUEST_QUEUE, does not become obsolete. See Chapter 3 for more information on generic completions and their effect on dependent units.

2.2.5 Using Units from Other Program Libraries

The program library manager allows you to use units from other program libraries either by direct copy or by reference.

The ACS COPY UNIT command allows you to copy one or more units into the current program library from another library. The ACS ENTER UNIT command allows you to create a reference from the current program library to units in another library. The process of entering references to units with the ACS ENTER UNIT command is called *entering units* (into the current program library from another library).

The choice of whether to copy or enter units depends on the circumstances, as described in the following sections. To use the ACS COPY UNIT or ENTER UNIT command, you must have read access to the program library

where the unit is stored (see Chapter 5 for more information on library access and library protection).

2.2.5.1 Copying Units into the Current Program Library

The ACS COPY UNIT command copies one or more units into the current program library from another library.

The following example shows the use of the ACS COPY UNIT command to copy the unit QUEUE_MANAGER from the program library DISK:[SMITH.SHARE.ADALIB] into the current program library:

```
$ ACS COPY UNIT DISK:[SMITH.SHARE.ADALIB] QUEUE_MANAGER
```

For each unit specified, the ACS COPY UNIT command copies the specification and body. Units that have been loaded with the ACS LOAD command or converted with the ACS CONVERT LIBRARY command, but not yet recompiled, are not copied.

When a unit is copied, information about the external source file associated with the unit is also copied. This information may affect the behavior of any subsequent ACS COMPILE commands, if you change the location of the external source file. Thus, you may need to manage the behavior of the ACS COMPILE command by taking one of the following actions:

- Using the ACS SET SOURCE command to direct the ACS COMPILE command to the correct location
- Using a concealed logical name to refer to the directory containing the source files and change the meaning of the logical name as necessary. See Chapter 5 for more information on concealed logical names.

Copied units behave and can be handled as if they had been compiled directly into the current program library. The ACS COPY UNIT command has no effect on the program library from which a unit has been copied.

Once a unit has been copied, it is independent of the unit from which it was copied. The same is not true for a unit that has been entered (see Section 2.2.5.2). Therefore, if the external unit you need is subject to frequent unexpected changes, you may want to use the ACS COPY UNIT command, rather than the ACS ENTER UNIT command, to create a stable local copy and minimize the impact on dependent units. However, when you use the ACS COPY UNIT command, you must keep track of when the original unit you copied has been modified.

If you find that the original unit has been revised and compiled again in its original program library, you can use the ACS COPY UNIT/REPLACE command to copy the modified version. If you use the ACS COPY UNIT command without the /REPLACE qualifier in this situation, the program library manager informs you that the unit already exists in the current program library and does not replace it.

If the unit you need to copy depends on other units, you can use the /CLOSURE qualifier to automatically copy the entire execution closure of the unit into the current program library. If the specified library is a sublibrary, then all parent libraries are searched for units in the closure.

For example, consider the following situation:

- The package QUEUE_MANAGER names the generic instantiation GUEST_QUEUE in a **with** clause.
- The generic instantiation GUEST_QUEUE depends on the generic package QUEUE.
- The units QUEUE_MANAGER, GUEST_QUEUE, and QUEUE have all been compiled into the program library DISK:[SMITH.SHARE.ADALIB].

The closure of the unit QUEUE_MANAGER includes the units QUEUE_MANAGER, GUEST_QUEUE, and QUEUE. The following command copies all of these units into the current program library:

```
$ ACS COPY UNIT/LOG/CLOSURE DISK:[SMITH.SHARE.ADALIB] QUEUE_MANAGER
%I, Generic instantiation GUEST_QUEUE copied
%I, Generic package QUEUE copied
%I, Generic package body QUEUE copied
%I, Package specification QUEUE_MANAGER copied
%I, Package body QUEUE_MANAGER copied
```

Note that the ACS COPY UNIT command makes local copies of units that have been entered into a given library (see Section 2.2.5.2), as well as units that have been compiled into a given library. Thus, the result of this example would have been the same if the unit QUEUE had been entered into DISK:[SMITH.SHARE.ADALIB] from yet another library, such as USER:[PROJECT.ADALIB].

2.2.5.2 Entering Units into the Current Program Library

The ACS ENTER UNIT command creates a reference in the current program library to a unit in another library. Units that have been loaded with the ACS LOAD command or converted with the ACS CONVERT LIBRARY command, but not yet recompiled, are not entered.

NOTE

The use of concealed-device logical names and rooted directory syntax to specify program libraries helps in working with entered units. See Chapter 5 for more information.

The following example shows the use of the ACS ENTER UNIT command to enter the unit QUEUE_MANAGER into the current program library from DISK:[SMITH.SHARE.ADALIB]:

```
$ ACS ENTER UNIT DISK:[SMITH.SHARE.ADALIB] QUEUE_MANAGER
```

For each unit specified, the ACS ENTER UNIT command enters a reference to the specification and a reference to the body. The ACS ENTER UNIT command has no effect on the library from which units have been entered.

You can determine which units are entered in the current program library by using the ACS DIRECTORY command. For example:

```
$STANDARD
    package specification          17-Feb-1983 00:00  <entered>
    .
    .
    .
AUX_IO_EXCEPTIONS
    package specification          13-Apr-1989 23:36  <entered>

CALENDAR
    package specification          13-Apr-1989 23:51  <entered>
    package body                   13-Apr-1989 23:52  <entered>

CDD_TYPES
    package specification          13-Apr-1989 23:52  <entered>
    .
    .
    .
```

You can also identify entered units by using the ACS SHOW PROGRAM command.

An example of entered units is the set of VAX Ada predefined units (STANDARD, SYSTEM, TEXT_IO, STARLET, and so on) that are entered into any newly created program library. The predefined units are entered from the program library on your system denoted by the logical name ADA$PREDEFINED.

If an entered unit is subsequently compiled in its original program library, any reference to that unit from another library is made obsolete. You cannot use the entered unit until you have *reentered* it using the ACS ENTER UNIT/REPLACE or ACS REENTER command. In contrast, compiling a unit that has been copied has no effect on the copies. Therefore, you may

want to use the ACS COPY UNIT command rather than the ACS ENTER UNIT command if the external unit is subject to frequent changes; see Section 2.2.5.1.

The ACS ENTER UNIT command is particularly useful for units that need to be used by several program libraries. You may want to share units for two reasons:

- Maintaining one master copy of a shared unit (or a set of shared units) conserves disk space.
- If an entered unit is modified and compiled again in its original library, all references to that unit from other libraries are made obsolete (the program library manager issues appropriate messages when you try to use the entered unit). Thus, you are assured that a revision to an entered unit is automatically detected in all libraries that share that unit.

The ACS CHECK, COMPILE, LINK, and RECOMPILE commands automatically warn you of any obsolete references to units that have been entered into the current program library. For example, consider the following situation:

- The main program, HOTEL, depends on the package RESERVATIONS.
- The specification of RESERVATIONS depends on the package SCREEN_IO, which has been entered from the library USER:[PROJECT.ADALIB].
- The body of RESERVATIONS depends on the package QUEUE_MANAGER, which has been entered from the library DISK:[SMITH.SHARE.ADALIB].
- Both SCREEN_IO and QUEUE_MANAGER have been modified and compiled more recently than HOTEL and RESERVATIONS.

When the main program HOTEL is linked, the program library manager issues the following messages:

```
$ ACS LINK HOTEL
%E, package specification SCREEN_IO has been recompiled in
     USER:[PROJECT.ADALIB] and must be reentered
%E, package body SCREEN_IO has been recompiled in
     USER:[PROJECT.ADALIB] and must be reentered
%E, package specification QUEUE_MANAGER has been recompiled in
     DISK:[SMITH.SHARE.ADALIB] and must be reentered
%E, package body QUEUE_MANAGER has been recompiled in
     DISK:[SMITH.SHARE.ADALIB] and must be reentered
```

These messages identify the entered units that need to be reentered to make their references current and usable. These units must be reentered before the obsolete dependent units in the current program library can be recompiled.

You can reenter units using either the ACS ENTER UNIT/REPLACE command or the ACS REENTER command. Use the ACS ENTER UNIT/REPLACE command when you need to reenter one or more units from one library; use the ACS REENTER command when you need to reenter a number of units from a number of libraries.

For example, you can use the ACS REENTER command with the asterisk wildcard character (*) to make current all obsolete references in your current program library, regardless of whether or not the references are to more than one other library:

```
$ ACS REENTER/LOG *

%I, Package specification $STANDARD entered
   .
   .
   .
%I, Package specification AUX_IO_EXCEPTIONS entered
%I, Package specification CALENDAR entered
%I, Package body CALENDAR entered
%I, Package specification CDD_TYPES entered
%I, Package specification CLI entered
%I, Package specification CONDITION_HANDLING entered
%I, Package specification CONTROL_C_INTERCEPTION entered
%I, Generic package DIRECT_IO entered
%I, Generic package body DIRECT_IO entered
%I, Package specification DIRECT_MIXED_IO entered
%I, Package body DIRECT_MIXED_IO entered
   .
   .
   .
%I, Package specification QUEUE_MANAGER entered
%I, Package body QUEUE_MANAGER entered
%I, Package specification SCREEN_IO entered
%I, Package body SCREEN_IO entered
   .
   .
   .
```

The units reentered in this example are from the libraries ADA$PREDEFINED, USER:[PROJECT.ADALIB], and DISK:[SMITH.SHARE.ADALIB].

After the obsolete entered units have been reentered, the remaining obsolete units can be recompiled in the current program library, using the ACS RECOMPILE command. By specifying HOTEL as the parameter to the ACS RECOMPILE command, all obsolete units in the closure of HOTEL are

recompiled. (See Chapter 3 for more information on recompilation and the ACS RECOMPILE command.)

```
$ ACS RECOMPILE HOTEL
```

The program HOTEL can now be linked.

2.2.6 Introducing Foreign (Non-Ada) Code into a Library

When you are working with mixed-language programs, you can use the ACS COPY FOREIGN and ENTER FOREIGN commands to introduce linkable non-Ada code into the current program library. You can then use ACS commands to manipulate the resulting units as though they were VAX Ada units.

The ACS COPY FOREIGN command copies a foreign object file into the current program library. The ACS ENTER FOREIGN command enters a reference to an external file into the current program library. An entered foreign file may be an object file, object library, shareable image library, shareable image, or linker options file. The /LIBRARY, /OBJECT, /OPTIONS, and /SHAREABLE qualifiers to the ACS ENTER FOREIGN command specify the kind of file you are entering; the default is an object file.

Before copying or entering a foreign file, you must create an Ada specification for it and compile that specification into the library. You then copy or enter the foreign file as a library body—that is, the body of a library package specification, library procedure specification, or library function specification. Note that compiling the specification of a unit that has a foreign body does not cause the body to become obsolete.

When you write a subprogram (procedure or function) specification that will have a foreign body, you must use the pragma INTERFACE and (optionally) a VAX Ada import pragma. See Chapter 4 for examples of linking; see the *VAX Ada Run-Time Reference Manual* for examples of writing mixed-language programs.

The ACS COPY FOREIGN and ENTER FOREIGN commands provide useful mechanisms for importing package bodies. In the following example, the body for IMPORTED_BODY is written in VAX Pascal. Note the use of the INITIALIZE attribute with the declaration of the Pascal procedure; without it the package body code is never "elaborated" and the variable Total never receives the value it is assigned in Procedure Pas_Body.

```
-- Ada package specification.
--
package IMPORTED_BODY is
   TOTAL: FLOAT;
   pragma IMPORT_OBJECT(TOTAL);
end IMPORTED_BODY;
------------------------------------------------------------
{ Pascal body. }

Module Pas_Body;
   VAR
      Total: [GLOBAL]REAL;

[INITIALIZE] Procedure Pas_Body;
   CONST
      Rate = 0.06;
   VAR
      Amt, Tax: REAL;
   BEGIN
      Amt := 5.0;
      Tax := Amt * Rate;
      Total := Tax + Amt;
   END;

END.
------------------------------------------------------------
-- Ada main program.
--
with IMPORTED_BODY; use IMPORTED_BODY;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure PRINT_TOTAL is
begin
   PUT(Total);
end PRINT_TOTAL;
```

You would compile the Ada and Pascal code in this example using the VAX Ada and VAX Pascal compilers, and then you would either copy or enter the resulting Pascal object file into the current program library. For example:

```
$ ACS ENTER FOREIGN PAS_BODY IMPORTED_BODY
```

Now, because the Pascal module Pas_Body is known to the current program library as the body of the Ada package IMPORTED_BODY, the Ada procedure PRINT_TOTAL can be linked using the ACS LINK command. See Chapter 4 for more information on linking mixed-language programs.

2.2.7 Deleting Units from the Current Program Library

You enter the ACS DELETE UNIT command to delete one or more units from the current program library. For each unit name specified, the ACS DELETE UNIT command deletes the specification and body. For example:

```
$ ACS DELETE UNIT/LOG SCREEN_IO
%I, Package specification SCREEN_IO deleted
%I, Package body SCREEN_IO deleted
```

This command is used in the same way regardless of whether a unit was compiled, copied, or entered into the library. The ACS DELETE UNIT command operates only on the current program library and has no effect on any other library.

If you want to delete only the body of a specified unit, you can use the /BODY_ONLY qualifier with the ACS DELETE UNIT command. In this case, the specification is not deleted. Thus, you can use the /BODY_ONLY qualifier to delete a package body for a package that has been redefined so that it no longer needs a body. For example:

```
$ ACS DELETE UNIT/BODY_ONLY/LOG SCREEN_IO
%I, Package body SCREEN_IO deleted
$ ACS DIRECTORY SCREEN_IO
SCREEN_IO
    package specification          15-Apr-1988 16:19

Total of 1 unit.
```

If you want to delete one or more entered units, you can use the /ENTERED qualifier with the ACS DELETE UNIT command. For example, the following command deletes all of the units entered from the library [SMITH.SHARE.ADALIB]:

```
$ ACS DELETE UNIT/LOG/NOLOCAL/ENTERED=[SMITH.SHARE.ADALIB] *
%I, Package instantiation GUEST_QUEUE deleted
%I, Generic package QUEUE deleted
%I, Generic package body QUEUE deleted
%I, Package specification QUEUE_MANAGER deleted
%I, Package body QUEUE_MANAGER deleted
```

Note that in this case, the /NOLOCAL qualifier is also required to prevent the local (nonentered) units from also being deleted (/LOCAL is the default).

2.3 Using Program Sublibraries

Although a single program library is useful in many software project situations, it may prove unwieldy when used for a system with many components or many developers. For example, every time a compilation unit is compiled, it is redefined in its program library, and the previous versions are discarded. Any errors introduced during the modification immediately affect dependent units. Moreover, if the modified unit is a library specification, all dependent units must be recompiled. Program *sublibraries* alleviate these problems by allowing you to isolate a collection of units while they are being developed or maintained.

The following sections give more detail on how to use sublibraries. The techniques discussed in these sections can be used with a project of any size. See Chapter 5 for information related to choosing a particular sublibrary configuration.

2.3.1 Using ACS Commands with Program Sublibraries

When using ACS commands with sublibraries, note the following points:

- The ACS CHECK, COMPILE, COPY FOREIGN, ENTER FOREIGN, EXPORT, EXTRACT SOURCE, LINK, RECOMPILE, and SHOW PROGRAM commands search the entire library hierarchy, starting with the crurent program library and working up through its parents to the root or ancestor parent library, for all units specified as parameters to the command using names that do not involve wildcard characters.

  For units selected with names that have wildcard characters, only the current program library is searched. The ACS LINK/MAIN (the default) and EXPORT/MAIN commands do not accept names with wildcard characters. However, the ACS LINK/NOMAIN and EXPORT/NOMAIN (the default) do accept names with wildcard characters.
- The ACS COPY UNIT, DELETE UNIT, DIRECTORY, ENTER UNIT, MERGE, and REENTER commands search only the current program library for the specified units, irrespective of wildcards.
- The ACS CHECK, COMPILE, COPY UNIT/CLOSURE, ENTER UNIT/CLOSURE, EXPORT, LINK, RECOMPILE, and SHOW PROGRAM commands, which operate on the execution closure of the units specified, search the entire library hierarchy for all (other) units in the closure, regardless of whether one of the other units was in the closure of a unit specified with a name with wildcards or not.

- All commands that search the entire library hierarchy select units according to the panes-of-glass visibility conventions described at the beginning of this chapter.

For example, the following command will search only the current program library for units whose names match the wildcard patterns B* and C% (for example, B1, B2, and C3). It will search the entire library hierarchy for units A and D, as well as all other units in the execution closure of A, B1, B2, C3, and D.

```
$ ACS CHECK A, B*, C%, D
```

The following command will search only the current program library for A, D, and units whose names match the wildcard patterns B* and C%:

```
$ ACS DIRECTORY A, B*, C%, D
```

2.3.2 Creating a Nested Sublibrary Structure

By specifying a sublibrary to be a parent library, you can use the ACS CREATE SUBLIBRARY command to create a nested sublibrary structure (see Section 2.1.1). Nested sublibraries give you the flexibility of creating additional controlled subenvironments for modifying units. The following command lines represent the structure shown in Figure 2–1:

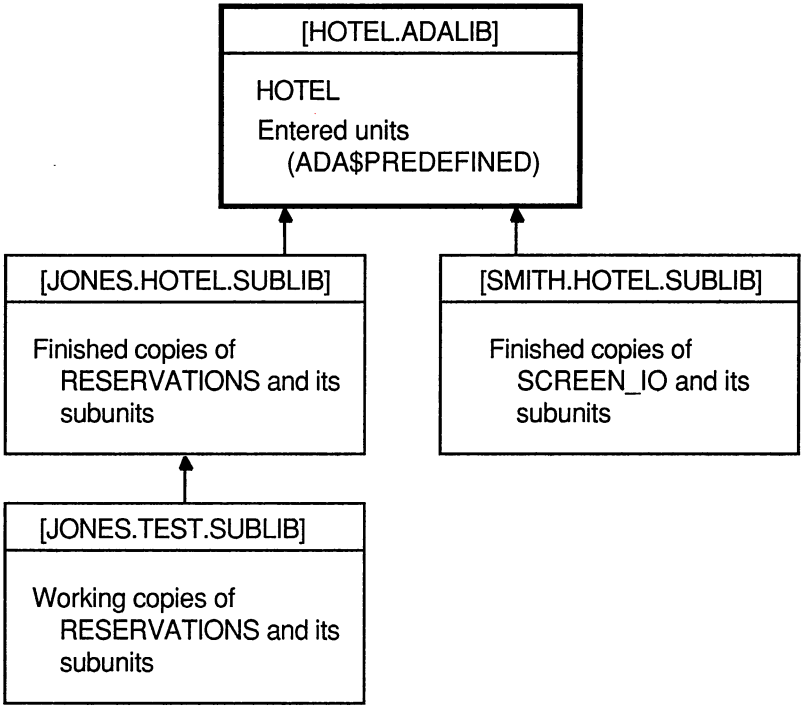
```
$ ACS CREATE SUBLIBRARY/PARENT=[HOTEL.ADALIB] [JONES.HOTEL.SUBLIB]

$ ACS CREATE SUBLIBRARY/PARENT=[JONES.HOTEL.SUBLIB] -
_$ [JONES.TEST.SUBLIB]
```

There is no specific limit on the depth of sublibrary nesting, but performance decreases as more sublibrary levels are added.

The ACS SHOW LIBRARY/FULL command identifies the immediate parent library of a sublibrary. For example:

```
$ ACS SHOW LIBRARY/FULL [JONES.HOTEL.SUBLIB]
Program library USER:[JONES.HOTEL.SUBLIB]

    Sublibrary
        of USER:[HOTEL.ADALIB]
   .
   .
   .
```

Figure 2–1: Simple Nested Sublibrary Structure



2.3.3 Changing the Parent of a Sublibrary

By using a concealed-device logical name and rooted directory syntax for the VMS specification of a parent library directory, you can later change the parent library. For example, the following command defines the root directory PROJECT_LIB to correspond to the device and directory DUA7:[PROJECT.ADALIB]:

```
$ DEFINE/TRANSLATION_ATTRIBUTES=CONCEALED -
_$ PROJECT_LIB DUA7:[PROJECT.ADALIB.]
```

The next command creates the sublibrary USER:[JONES.SUBLIB] with the parent PROJECT_LIB:[000000]. Note the use of [000000] to refer to the root directory, which in this case is [PROJECT.ADALIB].

```
$ ACS CREATE SUBLIBRARY/PARENT=PROJECT_LIB:[000000] -
_$ USER:[JONES.HOTEL.SUBLIB]
```

To change the parent library, all you need to do is redefine the logical name PROJECT_LIB. For example:

```
$ DEFINE/TRANSLATION_ATTRIBUTES=CONCEALED PROJECT_LIB -
_$ DUA6:[NEWPROJECT.ADALIB.]
```

For more information on using concealed-device logical names and rooted directory syntax with parent libraries and sublibraries, see Chapter 5. For general information on concealed-device logical names and rooted directory syntax, see the *VMS DCL Concepts Manual* and *Guide to VMS File Applications*.

2.3.4 Merging Modified Units into the Parent Library

The ACS MERGE command moves new versions of a set of units from a sublibrary into its parent library. By default, any earlier versions of the merged units are deleted from the parent library.

Units are not merged under the following circumstances:

- If they are older than the units in the parent library
- If a unit by the same name in the parent library has a more recent external source file
- If they have been loaded with the ACS LOAD command or converted with the ACS CONVERT LIBRARY command, but not yet recompiled

When a unit is merged, information about the external source file associated with the unit is also merged. This information may affect the behavior of any subsequent ACS COMPILE commands, if you change the location of the external source file. Thus, you may need to manage the behavior of the ACS COMPILE command by taking one of the following actions:

- Using the ACS SET SOURCE command to direct the ACS COMPILE command to the correct location.
- Using a concealed logical name to refer to the directory containing the source files and change the meaning of the logical name as necessary. See Chapter 5 for more information on concealed logical names.

The following command merges all of the units that are in the current program library (sublibrary) into the library's parent library:

```
$ ACS MERGE *
```

2.3.5 Modifying and Testing Units in a Sublibrary Environment

You modify and test units in a sublibrary environment by first isolating the units that need testing. Then, you generally follow these steps:

1. Create a sublibrary of the library where the units currently exist, and define the sublibrary to be the current program library.
2. Edit the source text for the units being modified. (Note that the VAX Ada program library manager does not provide a reservation system, so if you need to reserve the source files for the units you are modifying, you must use another tool such as the VAX DEC/Code Management System (CMS). See Chapter 5 for more information about managing modified source files.)
3. Compile the modified source text using the sublibrary as the current program library. Units are compiled in the context of both the sublibrary and its parent library, but only the sublibrary is updated with the compiled units. Therefore, the parent library remains stable while the units are being modified and tested independently in the sublibrary.
4. When you have finished modifying and testing the units, check the impact of any modifications with the ACS CHECK command. Then, use the ACS MERGE command to update the parent library with the latest versions of the modified units.
5. If you have used a mechanism such as CMS to reserve the source files, replace the source files.

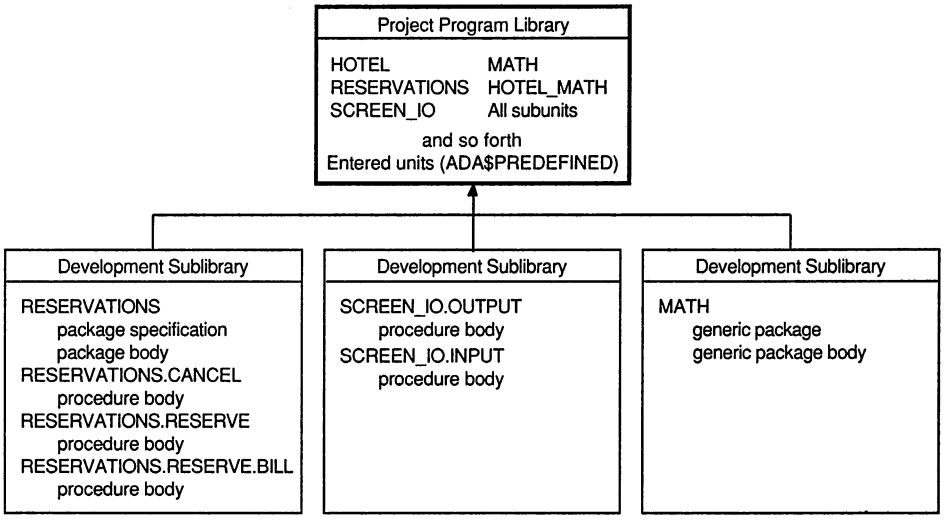
For example, consider the sublibrary configuration for the HOTEL program shown in Figure 2–2. Each of the sublibraries is to be used to modify and test a different set of units.

Modification and testing of the generic unit MATH would involve the following series of ACS commands:

1. Create the sublibrary and define it to be the current program library:

```
$ ACS CREATE SUBLIBRARY [JONES.HOTEL.SUBLIB]
$ ACS SET LIBRARY [JONES.HOTEL.SUBLIB]
```

Figure 2–2: Sublibrary Configuration for the HOTEL Program



2. Modify the source files in a working directory.
3. Compile the files into the sublibrary:

```
$ ADA MATH_, MATH
```

4. Enter the ACS CHECK command to determine the impact of the modifications on any dependent units in the parent library:

```
$ ACS CHECK HOTEL
%E, Obsolete library units are detected

%I, The following units need to be recompiled:
ACCOUNTING
    package instantiation          15-Apr-1989 16:35
RESERVATIONS
    package body                   15-Apr-1989 16:35
RESERVATIONS.RESERVE
    procedure body                 15-Apr-1989 16:35
RESERVATIONS.RESERVE.BILL
    procedure body                 15-Apr-1989 16:35
RESERVATIONS.CANCEL
    procedure body                 15-Apr-1989 16:35
```

By specifying the main program, you can detect obsolete units in the entire program.

5. Recompile any obsolete units:

```
$ ACS RECOMPILE HOTEL
```

   Note that only the sublibrary is updated when HOTEL is recompiled.

6. Link the entire program and run it to test its behavior:

```
$ ACS LINK HOTEL
$ RUN HOTEL
```

7. Repeat the previous steps, as necessary.
8. When the modified units are behaving correctly, merge them into the parent library:

```
$ ACS MERGE/LOG MATH
```

   See Section 2.3.4 for more information on merging units.

Because the sublibrary configuration in Figure 2–2 is set up so that each sublibrary contains a different set of units, you could execute the preceding steps concurrently for each sublibrary. However, because some units in one sublibrary may depend on units in another sublibrary, merging into the project program library needs to be coordinated carefully among project members. See Chapter 5 for additional information on configuring sublibrary structures and managing program development.

Chapter 3

Compiling and Recompiling VAX Ada Programs

In VAX Ada, compilation and recompilation are done in the context of the current program library, which can be either a program library or a sublibrary (see Chapter 2). Depending on the compilation command used, the source text to be compiled can come from two kinds of Ada source files:

- Files external to the library—source files edited and managed by you. These files are called *external source files*.
- Files internal to the library—files created by the /COPY_SOURCE compilation qualifier and managed by the program library manager. These files are called *copied source files*.

Each time a unit is compiled without error, the current program library is updated with the new unit and any other products of compilation, such as the object module and copied source file. If the compilation of the unit causes an error with a severity level greater than a warning (W), the current program library is not updated.

Whenever a unit is compiled, any dependent units are made obsolete and must be recompiled before the program can be linked. Linking also requires that all units in the execution closure (bodies and subunits) be current. Furthermore, any generic instantiations must be complete.

VAX Ada has four commands that you can use in different ways to compile, recompile, and complete units: the DCL ADA command, the ACS LOAD command, the ACS COMPILE command, and the ACS RECOMPILE command. Table 3–1 summarizes and compares the characteristics and use of each command.

See Chapter 1 for detailed definitions of obsolescence, currency, and incompletion.

Table 3–1: Summary Comparison of the DCL ADA and ACS LOAD, RECOMPILE, and COMPILE Commands

Command	Usage
($) ADA	Compiles the units in the specified Ada source files into the current program library. Useful for compiling units into a library for the first time or to compile again a set of units whose compilation order has changed. In both cases, you must know the compilation order. Available qualifiers provide a variety of options.
ACS LOAD	Processes the units in the specified Ada source files, and puts them into the current program library as obsolete units. You must recompile the units to make them current. Useful for putting a set of units into a library for the first time, especially if you do not know the compilation order. Also useful for adding units to an existing program. Available qualifiers are similar to the DCL ADA and ACS COMPILE and RECOMPILE qualifiers; has additional qualifiers to help select the files to be processed.
ACS RECOMPILE	Recompiles any obsolete unit or completes any incomplete generic instantiations in the execution closure of the specified units. Uses the copied source files stored in the program library. Ignores any source files external to the program library. Useful for making obsolete units current if the source files have not changed, for completing an incomplete generic instantiation, or for forcing the recompilation of an entire set of units with different qualifiers (such as /NOCHECK). For a unit to be recompiled or completed with this command, it must have been originally compiled with the /COPY_SOURCE qualifier. Available qualifiers are a superset of the qualifiers for the DCL ADA command, and are identical to the qualifiers for the ACS COMPILE command (although some qualifiers, like the /DIAGNOSTICS, /COPY_SOURCE, and /NOTE_SOURCE qualifiers, have no effect).

(continued on next page)

Table 3–1 (Cont.): Summary Comparison of the DCL ADA and ACS LOAD, RECOMPILE, and COMPILE Commands

Command	Usage
ACS COMPILE	Compiles any unit whose external source file has been modified, as well as recompiling obsolete units and completing incomplete generic instantiations in the execution closure of the specified units. To compile units whose source files have been modified, uses external source files (source files in the current default directory or source files in a location determined by a search list). To recompile obsolete units or complete incomplete generic instantiations, uses external source files if they are available; if external source files are not available, uses the copied source files stored in the program library. Useful for automatic compilation and recompilation of modified units whose external source files have changed. In cases where copied source files are used, units must have been compiled with the /COPY_SOURCE qualifier. You can use the ACS SET SOURCE command to specify the directories to be searched for the source files. Available qualifiers are a superset of the qualifiers for the DCL ADA command, and are identical to the qualifiers for the ACS RECOMPILE command.

The use of these commands is discussed in this chapter. The interpretation of compiler listings and diagnostic messages that may result from compilation are also discussed.

NOTE

The information in this chapter is task oriented. For full details on the format, parameters, and qualifiers of the various ACS commands, see Appendix A.

3.1 Compiling Units into a Program Library

To compile units into the current program library, you can use either the DCL ADA command or the ACS LOAD command. These two commands have different requirements and effects, as shown in Table 3–2.

Table 3–2: Comparison of the DCL ADA and ACS LOAD Commands

DCL ADA Command	ACS LOAD Command
Compiles the units contained in the files in the order given (or in the order within the file, if a file contains more than one unit).	Processes the units contained in the files; processing includes syntax checking and updating the library with unit dependence and source-file information, but the units are obsolete. The order in which the files are processed is not important.
Must be executed at DCL level; runs in interactive mode by default (unless executed in batch mode from a command procedure).	Can be executed at DCL or ACS level; runs in batch mode by default.
Takes one or more file specifications as parameters.	Takes one or more file specifications as parameters.
Cannot use wildcards in the file-specification parameters.	Can use wildcards in the file-specification parameters.
Must specify each file to be compiled.	Can use a number of qualifiers to select files based on backup and creation dates, user identification code, and so on.
Can specify the program library to be used for the duration of the compilation (/LIBRARY qualifier).	Cannot choose another program library.
When the command has finished executing, units compiled into the library are current and can be linked (assuming that their execution closure is complete).	When the command has finished executing, units loaded into the library are obsolete and must be recompiled with the ACS COMPILE or RECOMPILE command before they can be linked (see Section 3.2).
Best used in the following cases: • When you know the compilation order (for any number of units) • When fast compilation is important	Best used in the following cases: • When you do not know the compilation order of the units contained in the set of files; for example, after you fetch a CMS class of Ada files from a CMS library to build a system

In the following example, the ADA command compiles the two source files SCREEN_IO_.ADA and RESERVATIONS_.ADA in the order given. Because the default input file type for the ADA command is .ADA, the file type has been omitted in the command line.

```
$ ADA/LIST SCREEN_IO_,RESERVATIONS_
```

The /LIST qualifier causes a listing file (.LIS) to be created in the current default directory. In this example, one listing file is created for each of the two input files. The listing-file names are, by default, the same as the source-file names, but instead of a file type of .ADA, they have a file type of .LIS.

In the following example, the ACS LOAD command processes all of the units contained in the source files in the current default directory, and updates the current program library. Again, the default input file type is .ADA; the /LOG qualifier causes the files processed (not the units) to be listed.

```
$ ACS LOAD/LOG *
%I, The following files will be loaded:

DISK:[JONES.HOTEL]SCREEN_IO.ADA
DISK:[JONES.HOTEL]SCREEN_IO_.ADA
DISK:[JONES.HOTEL]RESERVATIONS.ADA
DISK:[JONES.HOTEL]RESERVATIONS_.ADA
DISK:[JONES.HOTEL]HOTEL.ADA

%I, Job ACS_LOAD (queue ALL_BATCH, entry 659) started
    on FAST_BATCH
```

The units are loaded into the library as obsolete units. To make them current and able to be linked, you must subsequently enter an ACS COMPILE or RECOMPILE command, keeping in mind that these commands operate on the execution closure of the units specified. For example:

```
$ ACS COMPILE HOTEL
```

See Section 3.2 for more information on recompiling obsolete units.

Both the DCL ADA and ACS LOAD commands accept more than one unit in a source file, but this practice is not recommended (see Chapter 1).

Both the DCL ADA and ACS LOAD commands assume the /COPY_SOURCE and /NOTE_SOURCE qualifiers by default. The /[NO]COPY_SOURCE qualifier controls whether copied source files are created in the current program library. Note the following points about this qualifier:

- When it is in effect, a copied source file is created in the current program library for each unit compiled without error.
- Copied source files are used by the ACS RECOMPILE and COMPILE commands.
- Copied source files are used by the debugger (see Chapter 6).

The /[NO]NOTE_SOURCE qualifier controls whether the compiler records the file specification of a unit's external source file in the program library. The ACS COMPILE command uses this information to locate revised source files. When it is in effect, the file specification of each unit's source file is recorded in the current program library when the unit is compiled without error.

Keep in mind that the default values of the /[NO]COPY_SOURCE and /[NO]NOTE_SOURCE qualifiers make the copied source files and the location of the source files available to anyone who has read access to a program library.

3.2 Recompiling Obsolete Units

Units can be obsolete for a number of reasons:

- One or more units that they depend on have been compiled more recently into the program library (see Chapter 1).
- The value of a global program library characteristic such as LONG_FLOAT or SYSTEM_NAME has been changed (for example, after you have used the ACS SET PRAGMA command). Note that the value of SYSTEM_NAME affects only those units that name the package SYSTEM in a **with** clause.
- The units were loaded into the current program library with the ACS LOAD command (see Section 3.1).

To recompile a set of obsolete units, you can enter either the ACS RECOMPILE or the ACS COMPILE command. In VAX Ada, the term *recompilation* refers to the following series of steps:

1. Formation of the execution closure of a given set of units
2. Identification of the obsolete units in the closure
3. Recompilation of the obsolete units

Table 3–3 notes the differences between the ACS RECOMPILE and COMPILE commands in performing these steps.

Table 3–3: Differences Between ACS RECOMPILE and COMPILE in Recompiling Obsolete Units

ACS RECOMPILE	ACS COMPILE
Performs only the recompilation steps	Performs the recompilation after compiling any units whose external source files have changed
Uses copied source files to do the recompilation	Uses external source files to do the recompilation; if external source files are not available, uses copied source files

Note the use of copied source files for recompilation. If a copied source file needed for a recompilation is missing (because /NOCOPY_SOURCE was specified in a previous compilation), the program library manager identifies the missing file, and the recompilation is not attempted. Thus, if you intend to use the ACS RECOMPILE command, you should not compile units with the /NOCOPY_SOURCE qualifier on any of the compilation commands.

The following example shows the use of the RECOMPILE command to recompile obsolete units. Consider the following set of units:

- The unit HOTEL, which is the main program and which names the unit RESERVATIONS in a **with** clause.
- The unit RESERVATIONS, whose specification names the unit SCREEN_IO in a **with** clause. The units RESERVATIONS and SCREEN_IO each have a specification, body, and some subunits.

All of the units have been compiled into the program library with the /COPY_SOURCE qualifier, so that a copied source file exists in the library for each unit.

If SCREEN_IO's specification is compiled again, then its dependent units become obsolete, as follows:

- The specification of RESERVATIONS becomes obsolete because it names SCREEN_IO in a **with** clause.
- The body of RESERVATIONS becomes obsolete because it depends on the specification of RESERVATIONS.
- The subunits of RESERVATIONS become obsolete because they depend on the body of RESERVATIONS.
- The unit HOTEL becomes obsolete because it names the unit RESERVATIONS in a **with** clause.

The following RECOMPILE command operates on the closure of RESERVATIONS, and compiles the obsolete units using the copied source files in the current program library:

```
$ ACS RECOMPILE/LOG RESERVATIONS

%I, The following units will be recompiled:
RESERVATIONS
    package specification           15-Apr-1989 16:35
    package body                    15-Apr-1989 16:35
SCREEN_IO
    package body                    15-Apr-1989 16:35
SCREEN_IO.INPUT
    procedure body                  15-Apr-1989 16:35
SCREEN_IO.INPUT.BUFFER
    function body                   15-Apr-1989 16:35
SCREEN_IO.OUTPUT
    procedure body                  15-Apr-1989 16:35
RESERVATIONS.RESERVE
    procedure body                  15-Apr-1989 16:35
RESERVATIONS.RESERVE.BILL
    procedure body                  15-Apr-1989 16:35
RESERVATIONS.CANCEL
    procedure body                  15-Apr-1989 16:35

%I, Job RESERVATIONS (queue ALL_BATCH, entry 180) started
    on FAST_BATCH
```

As shown in this example, you can use the /LOG qualifier to display the obsolete units in the order in which they will be submitted to the compiler.

The equivalent ACS COMPILE command would recompile the obsolete units using the external source files (as well as compiling any units whose source files had been modified); it would use copied source files only if the external files were not available. If any of the source files have been modified and the modifications change the order of compilation, the /PRELOAD qualifier is needed (see Section 3.4).

Note that the execution closure of a given unit does not include any units that name the given unit in a **with** clause. Therefore, in the preceding example, the unit HOTEL is not recompiled (although it is also obsolete) because HOTEL is not part of the execution closure of RESERVATIONS. If you were to specify HOTEL with the ACS RECOMPILE or COMPILE command, you would recompile the execution closure of HOTEL, which includes HOTEL, RESERVATIONS, and SCREEN_IO, and any subunits. Thus, to recompile the obsolete units of an entire program, you must specify the unit name of the main program with the RECOMPILE or COMPILE command.

Also note that the RECOMPILE and COMPILE commands do not recompile any entered units. However, because they check the execution closure of the units specified, these commands do detect obsolete units. For example:

```
$ ACS RECOMPILE HOTEL
%E, Package specification QUEUE_MANAGER has been recompiled in
        USER:[JONES.HOTEL.ADALIB] and must be reentered
%E, Package body QUEUE_MANAGER has been recompiled in
        USER:[JONES.HOTEL.ADALIB] and must be reentered
```

3.3 Completing Incomplete Generic Instantiations

An Ada program is considered to be incomplete if more processing needs to be done before the program can be linked. For example, a program with missing subunits is incomplete—you must compile the subunits into the program library before you can link the program. A program with incomplete generic instantiations is also incomplete—you must complete the instantiations before you can link the program.

An incomplete generic instantiation can occur for a number of reasons:

- If the body or subunits of the body for the corresponding generic unit are not available when the instantiation of the generic unit is compiled (in this case, you must compile the body or subunits before you can complete the instantiation). A special case of this situation occurs when the generic body is the result of another instantiation that has not been completed.
- If the body for the corresponding generic unit is compiled or recompiled after the instantiation of the generic unit is compiled.

You can use either the ACS RECOMPILE or the ACS COMPILE command to complete generic instantiations. The ACS RECOMPILE command uses copied source files to complete generic instantiations. The ACS COMPILE command uses external source files if they are available; copied source files if external source files are not available. In some cases, particularly when a generic unit contains an instantiation of another generic unit, you may need to use the ACS RECOMPILE or COMPILE command more than once to complete all of the instantiations in a set of units.

Note that when completing a generic instantiation, the compiler uses the values of the /CHECK, /DEBUG, and /OPTIMIZE qualifiers that were in effect when the instantiation was created. The compiler uses the original qualifier values even if you specify other values for the ACS RECOMPILE or COMPILE command that will perform the completion.

By default, a unit that contains a generic instantiation does not depend on the body for the corresponding generic unit. Thus, when the generic body is compiled, the unit containing the instantiation does not become obsolete, even though the instantiation has become incomplete. Consequently, when the unit containing the instantiation is recompiled to complete the instantiation, units that depend on the unit containing the instantiation do not become obsolete and do not need to be recompiled.

However, an implicit or explicit inline pragma for the generic instantiation may cause the unit containing the instantiation to depend on the body for the corresponding generic unit. If this dependence exists (see Chapter 1), the instantiation is expanded inline and the unit containing the instantiation becomes obsolete when the generic body is recompiled. Consequently, when the unit containing the instantiation is recompiled to complete the instantiation, the unit containing the instantiation is also recompiled, and all units that depend on the containing unit must also be recompiled.

See the *VAX Ada Language Reference Manual* and *VAX Ada Run-Time Reference Manual* for more information on inline pragmas (pragma INLINE and pragma INLINE_GENERIC). See Appendix A for information on the /OPTIMIZE qualifier, which has options that have effects equivalent to the inline pragmas. See Chapter 1 for more information on incomplete and obsolete units.

Consider the following set of units:

- The unit MATH is a generic package, with a specification (MATH_) and a body (MATH).
- ACCOUNTING is a library package instantiation of the unit MATH.
- The unit RESERVATIONS is a nongeneric package; its body depends on the package ACCOUNTING.
- The main program HOTEL depends on the unit RESERVATIONS.

The specifications and bodies of these units are compiled into the program library in the following order. Note that the body of the generic unit MATH is compiled after the instantiation ACCOUNTING.

```
MATH_
ACCOUNTING
RESERVATIONS_
RESERVATIONS
HOTEL
MATH
```

As the following commands show, the main program HOTEL cannot be linked until the body of the unit MATH is in the program library and ACCOUNTING has been completed:

```
$ ADA MATH_
$ ADA ACCOUNTING
$ ADA RESERVATIONS_, RESERVATIONS
$ ADA HOTEL
$ ACS LINK HOTEL
%E, Body for MATH not found in library
%E, Obsolete library units are detected

%I, The following units need to be completed (use ACS COMPILE
    or ACS RECOMPILE):
ACCOUNTING
    package instantiation           15-Apr-1989 16:35

$ ADA MATH
$ ACS LINK HOTEL
%E, Obsolete library units are detected

%I, The following units need to be completed (use ACS COMPILE
    or ACS RECOMPILE):
ACCOUNTING
    package instantiation           15-Apr-1989 16:35

$ ACS RECOMPILE/LOG ACCOUNTING
%I, The following units will be completed:
ACCOUNTING
    package instantiation           15-Apr-1989 16:35

%I, Job ACCOUNTING (queue ALL_BATCH, entry 383) started on
        FAST_BATCH

$ ACS LINK HOTEL
$ RUN HOTEL
```

Consider also the following example:

```
generic
package GENERIC_PACKAGE is
   procedure INNER_PROCEDURE;
end GENERIC_PACKAGE;

-------------------------------------------------------

with GENERIC_PACKAGE;
package CONTAINS_INST is
   package NEW_GENERIC_PACKAGE is new GENERIC_PACKAGE;
    . . .
end CONTAINS_INST;
```

```
-------------------------------------------------------

with CONTAINS_INST;
procedure MAIN is
begin
    . . .
end MAIN;

-------------------------------------------------------

package body GENERIC_PACKAGE is
   procedure INNER_PROCEDURE is
   begin
       . . .
   end INNER_PROCEDURE;
end GENERIC_PACKAGE;
```

Assume that the units are compiled in the order shown and that all compile without errors. Because the package body for GENERIC_PACKAGE is compiled after the package CONTAINS_INST, the instantiation of NEW_GENERIC_PACKAGE is incomplete. By entering an ACS COMPILE or RECOMPILE command, you would complete the instantiation.

You can detect incomplete generic instantiations during a compilation by checking the compiler listing file, or by using the /WARNINGS=(STATUS:TERMINAL) qualifier on your compilation command. For example:

```
$ ADA/WARNINGS=(STATUS:TERMINAL) GENERIC_PACKAGE

%I, Generic package GENERIC_PACKAGE added to library
        USER:[JONES.HOTEL.ADALIB]
    Replaces older version compiled  8-Mar-1989 21:15
%I, Package specification CONTAINS_INST added to library
        USER:[JONES.HOTEL.ADALIB]

   10       package NEW_GENERIC_PACKAGE is new GENERIC_PACKAGE;
...........1
%I, (1) Instantiation incomplete because the generic body
        for generic package GENERIC_PACKAGE in GENERIC_PACKAGE
        at line 1 is not available
%I, Procedure body MAIN added to library
        USER:[JONES.HOTEL.ADALIB]
%I, Generic package body GENERIC_PACKAGE added to library
        USER:[JONES.HOTEL.ADALIB]
    Replaces older version compiled  8-Mar-1989 21:15
    Corresponds to generic package GENERIC_PACKAGE compiled
        8-Mar-1989 21:16
 .
 .
 .
```

The ACS CHECK and SHOW PROGRAM commands also detect incomplete instantiations (see Chapter 2).

3.4 Compiling a Modified Program

To compile a modified program, you can use the ACS COMPILE command with one or more qualifiers and the unit name of the program. The COMPILE command locates modified source files, compiles them, and then recompiles any obsolete units using information stored in the program library from previous compilations. It also forms any generic completions involved in the compilation. For example:

- To locate the modified source files, the COMPILE command uses information obtained with the /NOTE_SOURCE compilation qualifier.
- To carry out recompilations and generic completions, it uses information obtained with the /NOTE_SOURCE qualifier; if it cannot find that information, it uses information obtained with the /COPY_SOURCE qualifier.

When the COMPILE command searches for modified source files, it searches source-file directories as indicated in Section 3.6. If the COMPILE command finds that no files have been modified and all units are current and complete, the program library manager issues a success message. For example:

```
$ ACS COMPILE QUEUE_MANAGER, ACCOUNTING
%I, All units and files current, no compilations required
```

If the COMPILE command cannot find the files it needs for compilation, recompilation, or to complete a generic instantiation, an error message is issued, and no compilation occurs. See Section 3.2 for more information on how the COMPILE command recompiles obsolete units. See Section 3.3 for more information on generic completions.

The following example shows the functions of the COMPILE command when it finds revised source files. The COMPILE command in this example was issued after the specification and body of RESERVATIONS were revised, but before they were compiled into the current program library. The command operates on the closure of RESERVATIONS, the specified unit. The /LOG qualifier displays the units to be compiled from external source files, and those to be recompiled either from external or copied source files.

```
$ ACS COMPILE/LOG RESERVATIONS

%I, The following units will be compiled from source files
RESERVATIONS
    package specification           15-Apr-1989 17:27
        USER:[JONES.HOTEL]RESERVATIONS_.ADA
    package body                    15-Apr-1989 17:27
        USER:[JONES.HOTEL]RESERVATIONS.ADA
```

```
%I, The following units will be recompiled
RESERVATIONS.RESERVE
    procedure body                  15-Apr-1989 17:21
RESERVATIONS.RESERVE.BILL
    procedure body                  15-Apr-1989 17:21
RESERVATIONS.CANCEL
    procedure body                  15-Apr-1989 17:21

%I, Job RESERVATIONS (queue ALL_BATCH, entry 218) started
    on FAST_BATCH
```

By default, the ACS COMPILE command does not look within a source file to determine the use of **with** clauses, subunit stubs, and so on when it does a compilation. Instead, it assumes that unit dependences have not changed.

However, the /PRELOAD qualifier does allow you to use the COMPILE command to compile a modified set of units whose compilation order has changed (or to which new units have been added). For example, if the compilation order of the units comprising the hotel reservation system has changed, then the following command will detect such changes and put the units in the correct order before submitting them to the compiler:

```
$ ACS COMPILE/PRELOAD HOTEL
```

Note that the /PRELOAD qualifier operation is done immediately, before the total set of compilations and recompilations is submitted to the batch queue.

3.5 Forcing the Compilation or Recompilation of a Set of Units

In some cases, you may want to force the compilation or recompilation of a set of units. For example, you may want to compile or recompile a set of units with different qualifier values, such as /NOOPTIMIZE or /NOCHECK. Because the ACS COMPILE and RECOMPILE commands use date-checking to determine which units to compile or recompile, you can use the /NODATE_CHECK qualifier to accomplish this task.

For example, the following command forces the recompilation of the specification, body, and subunits of RESERVATIONS with the /NOOPTIMIZE qualifier:

```
$ ACS RECOMPILE/NODATE_CHECK/NOOPTIMIZE RESERVATIONS
```

You can force the compilation or recompilation of an entire closure (except for entered units) by combining the /NODATE_CHECK qualifier with the /CLOSURE qualifier. For example, the following command compiles any modified units and forces the compilation of all other units in the HOTEL program with the /NOCHECK qualifier:

```
$ ACS COMPILE/NODATE_CHECK/CLOSURE/NOCHECK HOTEL
```

Because they contain most of the executable code, bodies and subunits are apt to be modified and compiled more often than specifications. You can use the /FORCE_BODY qualifier if a unit is current, but you want to force the compilation or recompilation of only its body (and, because they depend on the body, its subunits). In this way, any unit that depends on the specification by way of a **with** clause is not made obsolete. In the following example, the RECOMPILE command forces the recompilation of the body (and any subunits) of SCREEN_IO:

```
$ ACS RECOMPILE SCREEN_IO/FORCE_BODY, RESERVATIONS
```

The /NODATE_CHECK and /FORCE_BODY qualifiers are positional, and affect only the parameter for which they are specified. Therefore, the following command lines are not equivalent:

```
$ ACS RECOMPILE/NODATE_CHECK/NOOPTIMIZE SCREEN_IO,RESERVATIONS
$ ACS RECOMPILE/NOOPTIMIZE SCREEN_IO,RESERVATIONS/NODATE_CHECK
```

In the first command line, the specifications, bodies, and subunits of RESERVATIONS and SCREEN_IO are recompiled with the /NOOPTIMIZE qualifier. In the second command line, the specification, body, and subunits of RESERVATIONS are recompiled with the /NOOPTIMIZE qualifier, but only those units in the execution closure of SCREEN_IO that are obsolete are recompiled with the /NOOPTIMIZE qualifier.

3.6 Using Search Lists for External Source Files

The ACS SET SOURCE command allows you to define a search list for the ACS COMPILE command. Then, when it searches for an external source file, the COMPILE command first tries to use the source-file-directory search list defined with the most recent SET SOURCE command. If no SET SOURCE command has been entered for the current process, the default source-file-directory search order is as follows:

1. SYS$DISK:[ ] (the current default directory)
2. ;0 (the directory that contained the file when it was last compiled), or node::;0 (if the file specification of the source file being compiled contains a node name)

The search order takes precedence over the version number or revision date-time if different versions of a file exist in two or more directories. Within any one directory, the version of a particular file that has the highest number is considered for compilation.

One possible use of the ACS SET SOURCE command is to define a search list that includes a VAX DEC/CMS library; see Chapter 5 for more information on the interaction between CMS and the VAX Ada program library manager.

The following example shows the use of the ACS SET SOURCE command:

```
$ ACS SET SOURCE SYS$DISK:[],USER:[JONES.HOTEL],;0
```

After this command is executed, a subsequent ACS COMPILE command will search for source files first in the current default directory, then in USER:[JONES.HOTEL], then in the directory where a particular source file was last compiled.

The ACS SET SOURCE command assigns the specified search list to the process logical name ADA$SOURCE. The search list defined by the SET SOURCE command stays in effect until you either enter another SET SOURCE command or log out.

You can use the ACS SHOW SOURCE command to display the current search list selected by the last ACS SET SOURCE command. For example:

```
$  ACS SHOW SOURCE
%I, Current source search list (ADA$SOURCE) is
        SYS$DISK:[]
        USER:[JONES.HOTEL]
        ;0
```

3.7 Choosing Optimization Options

The /OPTIMIZE qualifier to the DCL ADA and ACS COMPILE and RECOMPILE commands gives you a number of options for controlling the level of optimization applied to your program by the compiler. You can also use this qualifier and its options to override the behavior of the pragmas OPTIMIZE, INLINE, INLINE_GENERIC, and SHARE_GENERIC.

There are four primary options: TIME, SPACE, DEVELOPMENT, and NONE. There are two secondary options, INLINE and SHARE, which have a number of values and which can be used in combination with the four primary options, or can be used themselves as primary options. The compiler issues informational messages when the options you have chosen affect pragmas in your program. See Appendix A for a detailed description of each option and its values.

In general, you should use the DEVELOPMENT option during active development, and you should use the secondary options to tune the performance of production programs.

The following optimization options generally give the best overall results:

/OPTIMIZE=DEVELOPMENT	Programs under active development
/OPTIMIZE=INLINE:MAXIMAL	Production programs that do not make extensive use of generics, or that do make extensive use of generics, but explicitly specify a pragma SHARE_GENERIC for larger generics that are instantiated many times

Maximal inline expansion often results in programs that execute faster. However, you should not use maximal subprogram or generic inline expansion during active development because changes to subprogram or generic bodies that are expanded inline can cause many other units to need to be recompiled.

The following options are also of interest:

/OPTIMIZE=INLINE:SUBPROGRAMS	Generally provides the fastest running code on VMS systems and usually results in decreased code size as well.
/OPTIMIZE=INLINE:GENERICS	Results in maximal generic inline expansion and generally optimizes execution time. All generic instantiations (except for those to which an explicit pragma SHARE_GENERIC applies) are expanded inline at the point of instantiation if the generic body is available.
/OPTIMIZE=SHARE:MAXIMAL	Maximizes generic code sharing. This option optimizes space at the expense of execution time. Note, however, that sharing will not occur unless the code that is generated for one instance is similar to the code for another. You should not use the SHARE:MAXIMAL options when you are compiling all of the files in your program. You will obtain better results if you use the pragma SHARE_GENERIC or compile a portion of your program with this option.

See the *VAX Ada Run-Time Reference Manual* for more information on inline expansion (subprogram and generic) and generic code sharing. See the *VAX Ada Language Reference Manual* for more information on the pragmas OPTIMIZE, INLINE, INLINE_GENERIC, and SHARE_GENERIC.

3.8 Processing and Output Options

When you load, compile, and recompile Ada compilation units, you have a variety of processing and output options available to you. This section describes the following options:

- Batch processing. Batch mode, using a dedicated batch queue, is recommended with the DCL ADA and ACS LOAD, COMPILE, and RECOMPILE commands.
- Retaining, for future use, a DCL command file generated by the ACS LOAD, COMPILE, and RECOMPILE commands.
- Executing the ACS LOAD, COMPILE, or RECOMPILE compilations in a subprocess.
- Using certain defaults, symbols, and logical names for the ACS LOAD, COMPILE, and RECOMPILE commands.
- Directing ACS LOAD, COMPILE, and RECOMPILE command output to the terminal and to files.

See Appendix A for complete details on the qualifiers and defaults that control these options.

3.8.1 Executing Compilations in Batch Mode

In a multiuser environment, you can improve the use of machine time by executing the DCL ADA and ACS LOAD, COMPILE, and RECOMPILE commands in batch mode, using a dedicated batch queue.

The suggested batch-queue and SYSGEN parameters for best use of system resources during compilation are specified in Appendix E and in the *VAX Ada Installation Guide*. These parameters should be set by your system manager. The batch-queue parameters limit the number of concurrent batch jobs (and, therefore, compilations), and define an expanded value for the working set size.

You can submit DCL ADA compilations in batch mode using command procedures and the DCL SUBMIT command. The DCL SUBMIT command makes all of the DCL batch options available with the DCL ADA command. See the *VMS DCL Dictionary* for more information on these options.

Because the ACS LOAD, COMPILE, and RECOMPILE commands typically cause several units to be processed, the compilations for these commands are executed in batch mode by default. (This default mode is equivalent to using the /SUBMIT qualifier with these commands.) The LOAD, COMPILE, and RECOMPILE commands submit compilations to the batch queue named by the logical name ADA$BATCH by default. If ADA$BATCH is not defined, the system batch queue SYS$BATCH is used.

To use a dedicated queue for VAX Ada compilations, define ADA$BATCH as a logical name whose translation is the name of the appropriate queue. Consult your system manager for additional information.

3.8.2 Saving the Load or Compiler Command File

When you use the ACS LOAD, COMPILE, or RECOMPILE command, the program library manager creates a DCL command file. The file contains commands to load or compile units in appropriate order.

By default, the program library manager deletes the command file when the ACS LOAD, COMPILE, or RECOMPILE command is completed or terminated.

You can use the /COMMAND[=filespec] qualifier to retain the command file and optionally provide a file specification. When you use the /COMMAND qualifier, the program library manager does not perform the load operation or invoke the compiler. For example, the following command line creates the command file HOTEL.COM, which contains commands to compile every unit in the closure of HOTEL from source files:

```
$ ACS COMPILE/CLOSURE/NODATE_CHECK/COMMAND HOTEL
```

By default, the command file is created in the current default directory, and the file name is the name of the first unit specified (the default file type is .COM).

You can then edit the command file and later submit it as a batch job, using the DCL SUBMIT command. One possible use of this technique is to use certain batch qualifiers that are supported by DCL but not by the program library manager.

3.8.3 Loading Units and Executing Compilations in a Subprocess

You can cause the ACS LOAD, COMPILE, or RECOMPILE compilations to be executed in a subprocess by specifying the /WAIT qualifier. For example, the following command line creates a subprocess and invokes the compiler command file created by the program library manager to recompile the closure of unit RESERVATIONS:

```
$ ACS RECOMPILE/WAIT RESERVATIONS
```

The current process is suspended while the program library manager executes the command, and you must wait until the command is terminated before you can enter another command. The net effect is like executing the command interactively.

In a multiuser environment, you should execute the compilations for the ACS LOAD, COMPILE, and RECOMPILE commands in batch mode rather than in a subprocess (see Section 3.8.1).

3.8.4 Conventions for Defaults, Symbols, and Logical Names

When executing the ACS LOAD, COMPILE, or RECOMPILE command, the program library manager transmits the current definitions of certain defaults, symbols, and logical names to the batch or subprocess environment. Specifically:

- The current default directory is preserved. By default, any files created outside the current program library (for example, a command file or a listing file) are created in the current default directory.
- The current definition of the symbol ADA is used. For example, you could define ADA as follows:

  ```
  $ ADA == "ADA/LIST"
  ```

  Then the following commands would have the same effect:

  ```
  $ ACS COMPILE/NODATE_CHECK/NOOPTIMIZE SCREEN_IO
  $ ACS COMPILE/NODATE_CHECK/LIST/NOOPTIMIZE SCREEN_IO
  ```

- The current value of the logical name ADA$LIB is used to maintain the current program library context.

The DCL command file that you can obtain with the /COMMAND qualifier contains the current definitions of the default directory, the symbol ADA, and the logical name ADA$LIB.

3.8.5 Directing Program Library Manager and Compiler Output

When you use the ACS LOAD, COMPILE, or RECOMPILE command, any program library manager output and diagnostic messages generated before the compiler is invoked are directed to SYS$OUTPUT, by default. Examples of such ACS output and diagnostics include the following:

- A list of the units to be processed, as displayed by the /LOG qualifier
- A diagnostic message indicating that some units are obsolete or missing

You can use the /OUTPUT=file-spec qualifier to direct program library manager output and diagnostic messages to a file (in that case, program library manager diagnostic messages are directed to both the file and SYS$OUTPUT).

Diagnostic messages issued by the compiler are directed as follows:

- To a batch log file in the case of a batch job
- To your terminal in the case of a subprocess

The batch log file is created in your current default directory by default. You can use the /BATCH_LOG=file-spec qualifier with the ACS LOAD, COMPILE, and RECOMPILE commands to specify the target directory (and/or file name) for the batch log file.

3.9 Compiler Diagnostic Messages

When you compile an Ada program or compilation unit, you may receive a variety of diagnostic messages from the compiler. These messages are discussed in the following sections. All diagnostic messages issued by the VAX Ada compiler are listed and categorized in Appendix F.

NOTE

The DCL ADA and ACS command examples in this manual that involve diagnostic messages show only the severity part of the message code. They do not show the facility or the IDENT parts of the message code. To obtain this effect, use the following DCL SET MESSAGE command:

```
$ SET MESSAGE/NOFACILITY/NOIDENTIFICATION/SEVERITY/TEXT
```

To display or suppress various parts of diagnostic messages (including parts of the code) at the terminal or in a listing, enter other variants of the DCL SET MESSAGE command (see the *VMS DCL Dictionary* or *VMS General User's Manual*).

Diagnostic messages are also issued by the VAX Ada program library manager and run-time library; those messages are described, listed, and categorized in Appendixes G and H, respectively.

3.9.1 Diagnostic Messages and Their Severity

A VAX Ada compiler diagnostic message contains one of the following four codes, which indicate the severity level:

```
%F, message-text
%E, message-text
%W, message-text
%I, message-text
```

- **F** indicates a fatal error. The program library is not updated for the compilation unit in which the fatal error occurred. An F-level message indicates that the compiler is unable to perform the intended compilation. For example, the file to be compiled does not exist, or the library cannot be accessed. If an error is so serious that the compiler cannot continue, the entire compilation (not limited to the current compilation unit) is terminated with an F-level message that indicates the last line analyzed in the attempted compilation.
- **E** indicates a user error that makes the program illegal. The program library is not updated for the compilation unit in which the error occurred. E-level messages are often supplemented with informational (I-level) messages that give additional information about the error.

  When the VAX Ada compiler finds a syntax error, it attempts to correct it so that it can continue analyzing the rest of the program, if possible. A syntax error makes the program illegal, even if the temporary repair results in no further problems being uncovered.

  The compiler performs several kinds of local repairs. For example, it may add or delete a delimiter or reserved keyword. If local repair is considered inappropriate, the compiler may ignore the innermost declaration or statement. When a syntactically correct program results from these actions, processing continues with semantic analysis to provide as much useful diagnostic information as possible.

- **W** indicates a definite problem in a legal program—for example, an unknown pragma. The program library is updated for the compilation unit in which the warning occurred. A W-level error will not prevent the unit from linking and executing, but the behavior of the program may not be what you expect.
- **I** indicates an informational message. Section 3.9.2 describes the different kinds of informational messages and how you can control their display. An I-level message does not report an illegal construct as such. Frequently, however, the message contains supplementary information about a preceding or otherwise related E-level error. In addition, I-level messages are used to note places where some kind of exception (such as CONSTRAINT_ERROR) is likely to occur during execution, or to report that the compilation was successful and the program library has been updated.

When the compiler finishes or terminates a compilation, it exits with a status value that indicates the severity of the most severe error during execution. The status values and their severity are as follows:

Value	Severity
0	Warning
1	Success
2	Error
3	Informational
4	Fatal error

In VAX Ada, weak warnings fall under the category of informational, so keep the following points in mind:

- If the most severe error during execution of the image was a weak warning, then the compiler exits with a status that has a severity of informational (value 3).
- If no errors, warnings, or weak warnings are detected, the compiler exits with a status that has a severity of success (status value 1).

If you are running the compiler from a command procedure (batch), and need to check for weak warnings, such as one indicating that CONSTRAINT_ERROR will be raised at run time, you can include the following statement in your procedure:

```
$ IF $SEVERITY .EQ. 3 THEN  . . .
```

3.9.2 Informational Messages and the /[NO]WARNINGS Qualifier

There are four kinds of informational (I-level) messages:

- WEAK_WARNINGS indicate potential problems in a legal program—for example, a possible run-time error. Weak warnings are the only kind of informational diagnostics that are counted in the summary statistics given at the end of a compilation. The following is an example of a WEAK_WARNINGS message:

```
%I, CONSTRAINT_ERROR will be raised here if executed
```

- SUPPLEMENTAL messages are associated with a W-level or E-level diagnostic. Such messages provide additional information about a diagnostic or indicate that some checks were not performed due to prior errors. For example:

```
%I, Result type of expression is unknown due to prior error
```

- COMPILATION_NOTES provide information about how the compiler translated a program. They do not warn you of a possible problem, nor are they related to a W-level or E-level diagnostic. For example:

```
%I, Component allocated at ...

%I, Selected passing mechanism is ...

%I, Parent type chosen is ...

%I, Call of function X at line 2 is expanded inline ...
```

- STATUS diagnostics include some end-of-compilation statistics and other status messages. For example:

```
%I, Procedure body HOTEL added to program library
```

You can use the /WARNINGS=option qualifier on any of the VAX Ada compilation commands to control the display of I-level and W-level messages. The option specified with the /WARNINGS qualifier consists of a destination code for each kind of message. The possible code values are ALL, NONE, or any combination of TERMINAL (terminal device), LISTING (listing file), or DIAGNOSTICS (diagnostics file). See the compilation command descriptions (DCL ADA and ACS LOAD, COMPILE, and RECOMPILE) in Appendix A for the exact syntax. The defaults are as follows:

```
/WARNINGS=(NOCOMPILATION_NOTES, STATUS=LIST, SUPPLEMENTAL=ALL,
          WARNINGS=ALL, WEAK_WARNINGS=ALL)
```

For example, the following command specifies that weak warning and supplemental messages be sent to the terminal and to the listing file, and that other diagnostics be directed to their default destination:

```
$ ADA/LIST/WARN=(WEAK:(TERM,LIST),SUPP:(TERM,LIST)) SCREEN_IO.ADA
```

3.9.3 Setting Compiler Error Limits

You can use the /ERROR_LIMIT qualifier to control whether execution of the DCL ADA or ACS LOAD, COMPILE, or RECOMPILE command for a given compilation unit is terminated upon the occurrence of the nth E-level error within that unit.

Error counts are not accumulated across a sequence of compilation units. For example, if /ERROR_LIMIT=5 is specified, each compilation unit submitted may have up to four errors without terminating the compilation. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues.

The default value of the qualifier is /ERROR_LIMIT=30.

3.10 Compiler Listing Format

Example 3–1 shows various aspects of the listings generated by the VAX Ada compiler.

Example 3–1 is a listing from the compilation of a procedure (SHOW_LISTING) containing several kinds of diagnostic messages. The listing shows the kinds of error reporting provided by the VAX Ada compiler. The listing was the result of the following DCL ADA command:

```
$ ADA/LIST/SHOW=ALL/WARNINGS=COMPILATION_NOTES:LIST  LISTING
```

A VAX Ada compiler listing generated by the /LIST qualifier has, as a minimum, the following major sections:

- A source-code listing, showing any diagnostic messages at the appropriate locations, and an end-of-compilation status message
- A compilation-statistics section

When you use the /SHOW=PORTABILITY qualifier (the default), the listing includes a portability summary before the compilation statistics section.

When you use the /MACHINE_CODE qualifier, the listing includes a machine-code listing and a PSECT map after the source-code listing.

The following list notes various parts of a VAX Ada compiler listing. The numbers match identifying numbers in Example 3–1.

Each page of a compiler listing begins with a title line, which has five parts:

❶ The name of the compilation unit.

❷ The title text provided in a pragma TITLE, if any. If no title text applies, the compilation unit name may extend into this part. When title text does apply, the unit name is truncated to 15 characters, if necessary.

❸ The date and time of compilation.

❹ The VAX Ada compiler name and version number.

❺ The page number of the listing.

On each page of the listing, a line under the title line provides the following information about the source file:

❻ The module identifier, which was not used in the compilation for Example 3–1, and is specified by the default value 01.

❼ The subtitle text provided in a pragma TITLE, if any.

❽ The date and time of source file creation.

❾ The VMS file specification of the source file.

❿ The page number of the source file.

The source code is printed in the source-code listing as follows:

⓫ The compiler assigns a unique line number to each line of source code in a VAX Ada compilation unit. The symbolic traceback that is displayed if your program encounters an error at run time refers to these line numbers; in addition, the VMS Debugger uses these line numbers in various contexts (see Chapter 6).

The source-code listing includes diagnostic messages (errors, warnings, and informational messages) produced by the compiler. These messages appear directly after the line on which the condition is detected. In some cases, several lines may be devoted to one condition. Diagnostic messages include the following information:

⓬ A digit that points to the position on the line where the error or condition was detected.

⓭ A code that indicates the facility that issued the message, the severity of the diagnostic, and the IDENT (identification) of the diagnostic (for example, %ADAC-E-CL_UNINOTFOU). Note that only the severity part of the code (for example, E) is shown in Example 3–1.

⓮ A digit in parentheses, which corresponds to the digit that identifies the position where the error or condition was detected.

⓯ The text that corresponds to the particular condition detected.

Note that one source code error often causes other errors to be detected at the same position. Refer to Section 3.9 for more information on this subject.

The portability summary ⓰, provided by the (default) /SHOW=PORTABILI qualifier, identifies potentially nonportable features of the compilation unit.

The compilation-statistics section of the listing ⓱ includes the following information:

⓲ The exact command line passed to the VAX Ada compiler by the DCL ADA or ACS COMPILE or RECOMPILE command.

⓳ A list of the qualifier options in effect during the compilation.

⓴ Statistics regarding the internal timing of the compiler.

㉑ Diagnostic totals, by category, and page fault counts and timing totals.

Example 3–1: Sample VAX Ada Compiler Listing

```
LISTING ❶  Project DEMO ❷ 15-Apr-1989 17:53:14 ❸ VAX Ada X2.0-0 ❹    Page 1 ❺
01 ❻            Listing ❼ 15-Nov-1988 17:52:46 ❽ LISTING.ADA;2 ❾        (1) ❿
   ⓫
    1  pragma TITLE("Project DEMO", "Listing");
    2
    3  -- LISTING.ADA
    4  --
    5  -- A simple program to show the listing format and error
    6  -- reporting facilities of the VAX Ada compiler.
    7  --
    8   with NO_SUCH_UNIT; use NO_SUCH_UNIT;
.............1.................2 ⓬
⓭ ⓮ ⓯
%E, (1) Unit NO_SUCH_UNIT not found in library
%E, (2) NO_SUCH_UNIT is not declared [LRM 8.3]
```

(continued on next page)

Example 3–1 (Cont.): Sample VAX Ada Compiler Listing

```
    9  with SYSTEM;
   10  with TEXT_IO; use TEXT_IO;
   11  procedure LISTING is
   12
   13       OBJ1 : NO_SUCH_TYPE := 1;
.....................1.............2
%E, (1) NO_SUCH_TYPE is not declared [LRM 8.3]
%I, (2) Type checking is not complete; the type required from context is
        unknown due to a prior error

   14
   15       type E is (E1, E2, E3);
   16       OBJE : E;
............1
%I, (1) The representation of type E at line 15 is forced here

   17       for E use (-1, 5, 18);
.............1
%E, (1) The representation of type E at line 15 has already been forced at
        line 16 [LRM 13.1(6)]

   18
   19       pragma PACK(T);
.......................1
%W, (1) T is not declared in this declarative part; pragma PACK ignored
        [LRM 6.3.2(3), 12.1a, 12.1b, 13.1(5), 13.9(3)]

   20       pragma UNKNOWN_PRAGMA(T, T+1, OPTION => MAGIC_NUMBER-2);
.................1
%W, (1) Pragma UNKNOWN_PRAGMA is not known to this implementation; pragma
        ignored [LRM B]

   21
   22  begin
   23
   24       OBJ1 := -1
......................1
%E, (1) Inserted ";" at end of line

   25       OBJE := E'SUCC(F);
.........................1
%E, (1) F is not declared [LRM 8.3]

   26
   27  LOOPNAME1:
   28       loop
```

(continued on next page)

Example 3–1 (Cont.): Sample VAX Ada Compiler Listing

```
LISTING   Project ADADEMO  15-Apr-1989 17:53:14  VAX Ada 2.0-0     Page 2
01        Show Listing     15-Nov-1989 17:52:46  LISTING.ADA;2       (1)

   29           goto LOOPNAME2;
.....................1
%E, (1) Block or loop identifier LOOPNAME2 at line 32 is not a label

   30        end loop;
.....................1
%E, (1) Name LOOPNAME1 not specified at end of loop starting at line 27
        [LRM 5.5(3), 5.6(3)]

   31
   32   LOOPNAME2:
   33       for I in -1 .. 10 loop
.........................1
%E, (1) Type {universal_integer} is not allowed for the discrete range of a
        constrained array definition, an iteration rule, or an index of an
        entry family [LRM 3.6.1(2)]
%I, (1) Default resolution to type INTEGER does not apply because one or both
        expressions is not a literal, named number, or attribute; however,
        type INTEGER is assumed [LRM 3.6.1(2)]

   34           null;
   35       end loop LOOP2;
.....................1
%E, (1) Name LOOP2 does not match LOOPNAME2 at line 32 [LRM 5.5(3), 5.6(3),
        6.3(4), 7.1(3), 9.5(7)]

   36
   37  end;

PORTABILITY SUMMARY (16)

with SYSTEM                          9
enumeration representation clause
                                    17
pragma PACK                         19
unknown pragma(s)                   20
```

(continued on next page)

Example 3–1 (Cont.): Sample VAX Ada Compiler Listing

```
LISTING Project DEMO                    15-Apr-1989 17:53:14 VAX Ada 2.0-0 Page 3
01      Ada Compilation Statistics (17) 15-Nov-1989 17:52:46 LISTING.ADA;2     (1)

COMMAND QUALIFIERS (18)

  ADA/LIST/SHOW=ALL/WARN=COMP:LIST LISTING

QUALIFIERS USED (19)
    /NOANALYSIS_DATA/CHECK/COPY_SOURCE/DEBUG=ALL/ERROR_LIMIT=30/LIST
        /NOMACHINE_CODE
    /NODESIGN
    /NODIAGNOSTICS/LIBRARY=ADA$LIB
    /LOAD=REPLACE/NOTE_SOURCE/OPTIMIZE=(TIME,INLINE=NORMAL,SHARE=NORMAL)
              /SHOW=PORTABILITY/NOSYNTAX_ONLY
    /WARNINGS=(COMPILATION_NOTES=LIST,STATUS=LIST,SUPPLEMENTAL=ALL,WARNINGS=ALL,
              WEAK_WARNINGS=ALL)

COMPILER INTERNAL TIMING (20)

        Phase                          CPU      Elapsed    Page      I/O
                                     seconds    seconds    faults    count
  Initialization                       0.32       2.67       394       16
     Access ADALIB                     0.09       0.60        42        9
  Parser                               0.27       0.68       459       22
  Static semantics                     0.36       1.56       258       41
  Listing generation                   0.03       0.04        30        0
  Compilation library                  0.13       2.83        90       16
  Collect analysis data                0.00       0.00         0        0
  Compiler totals                      1.26       8.40      1308       97

COMPILATION STATISTICS (21)

  Weak warnings:             0
  Warnings:                  2
  Errors:                   10
  NYIs:                      0

  Peak working set:       3512
  Virtual pages used:    11084
  Virtual pages free:   163916
  CPU Time:              00:00:01.26   (1761 Lines/Minute)
  Elapsed Time:          00:00:08.40
  Compilation Complete
```

Chapter 4

Linking Programs

After you have compiled all of the units of your VAX Ada program, you must link the resulting object modules to form an executable image before you can run the program.

On the VMS operating system, linking is performed by the VMS Linker. To link VAX Ada object modules, you invoke the VMS Linker through the program library manager using the ACS LINK command (you do not invoke the VMS Linker directly). The ACS LINK command operates in the context of the current program library and performs the following steps:

1. Forms the execution closure of the main program.
2. Verifies that all units are defined in the current program library and are current. If any units are obsolete, incomplete, or missing, the command is terminated before the linker is invoked.
3. Creates an object file in the current default directory to elaborate any library packages in the closure at run time.
4. Creates a DCL command file that contains commands to invoke the VMS Linker to link all of the units.
5. By default, spawns a subprocess of your current process and invokes the linker command file just created (Section 4.3 describes the processing and output options available with the ACS LINK command). When the linker is invoked, it performs the following functions:
   - Combines object modules into one executable image
   - Resolves local and global symbolic references in the object code
   - Assigns values to global symbolic references
   - Generates an error message for any unresolved symbolic references
6. After the link operation is completed, deletes both the linker command file and the object file that was created to elaborate library packages.

Note that, as the VAX Ada interface to the linker, the program library manager performs several necessary operations before invoking the linker. Also, the ACS LINK command allows you to select several processing and output options through appropriate qualifiers.

The result of a successful link operation is an executable image. The default file specification for the image is as follows:

```
SYS$DISK:[]main-program-name.EXE
```

SYS$DISK is a system and/or process logical name that generally represents your default disk, and [] represents your current default directory, not your program library.

This chapter explains how to accomplish linking in the VAX Ada environment.

See Appendix A for more information on the ACS LINK command and its qualifiers.

4.1 Linking Programs Having Only VAX Ada Units

If your program consists only of VAX Ada units that are defined in the current program library or its parent library, enter the ACS LINK command with a single parameter: the name of the main program. For example:

```
$ ACS LINK HOTEL
```

This command causes the execution closure of HOTEL to be formed, and obsolete or incomplete units to be identified. If there are no obsolete or incomplete units, an object file and DCL command file are created, and the command file is executed to link all of the units in the closure. Finally, the image file HOTEL.EXE is created in the current default directory.

If the ACS LINK command does detect obsolete or incomplete units, you must recompile before the link operation will succeed. See Chapter 3 for more information on recompiling obsolete units and completing units containing incomplete generic instantiations.

4.2 Linking Mixed-Language Programs

The VAX Ada program library manager provides a number of link-related features that allow you to link Ada unit object modules with non-Ada object modules, as well as with object libraries and shareable image libraries. You can also use linker options files. These features are supported by the following ACS commands:

- The ACS LINK command syntax and qualifiers allow you to link Ada units directly against non-Ada object files, object libraries, shareable image libraries, or linker options files.
- The ACS COPY FOREIGN command allows you to copy a non-Ada object file into your current program library. You can then use the ACS LINK command to link the object file as the body for a library package or subprogram specification.
- The ACS ENTER FOREIGN command allows you to enter a reference to a non-Ada object file, object library, shareable image library, shareable image, or linker options file into your current program library. When you execute the ACS ENTER FOREIGN command, you associate the reference with a library package or subprogram specification. You can then use the ACS LINK command to link the reference as the body for the associated library package or subprogram specification.
- The ACS EXPORT command creates a concatenated object file for the closure of one or more VAX Ada units in your current program library, and places the file in your current default directory by default. You can then use the DCL LINK command to link the concatenated object file with non-Ada object files.

The following sections discuss the use of these features in more detail. See Appendix A for complete descriptions of the syntax and qualifiers for the ACS COPY FOREIGN, ENTER FOREIGN, and EXPORT commands.

4.2.1 Using the ACS COPY FOREIGN and ENTER FOREIGN Commands

The ACS COPY FOREIGN and ENTER FOREIGN commands allow you to introduce linkable non-Ada files into your program library. Foreign files that have been copied or entered into your program library in this manner are then handled by the ACS LINK command as Ada units.

When you use the ACS COPY FOREIGN or ENTER FOREIGN command, you copy or enter a foreign file as a library body—that is, the body of a library package specification, library procedure specification, or library function specification. Before you can copy or enter a foreign file, you must have compiled an Ada specification for it into the program library. The specification must contain the pragma INTERFACE and (if appropriate) a pragma IMPORT_FUNCTION, IMPORT_PROCEDURE, or IMPORT_VALUED_PROCEDURE for any procedure or function that the specification requires.

For example, consider the following situation:

- You have a VAX Ada procedure named ADA_CALLER that calls a squaring function named SQR.
- The body of SQR is written in VAX Pascal.

Before you can copy or enter the body of SQR into your program library, you must write a specification for SQR and compile it into the program library. For example, you could specify SQR as a library function whose body is to be imported:

```
-- Ada function specification for SQR
--
function SQR (Y : INTEGER) return INTEGER;
pragma INTERFACE (PASCAL, SQR);
pragma IMPORT_FUNCTION (INTERNAL => SQR,
                        EXTERNAL => SQUARE,
                        PARAMETER_TYPES => (INTEGER),
                        RESULT_TYPE     => INTEGER);
```

In this example, the EXTERNAL parameter in the pragma IMPORT_ FUNCTION indicates that SQUARE is the name of the Pascal routine that will serve as the body for the Ada function SQR. (See the *VAX Ada Run-Time Reference Manual* and *VAX Ada Language Reference Manual* for detailed information on the syntax for and use of the VAX Ada import pragmas.)

Assume that the Pascal routine SQUARE is coded as follows (note the use of the GLOBAL attribute):

```
{ Foreign (Pascal) function SQUARE }
MODULE SQUARE;
[GLOBAL] FUNCTION Square (X : Integer) : Integer;
   BEGIN
    . . .
   END;
END.
```

Also assume that the Ada procedure ADA_CALLER mentions SQR in a **with** clause:

```
with SQR;
procedure ADA_CALLER is
 . . .
end ADA_CALLER;
```

Then, you would use the following series of commands to create a library body from the foreign file (the default file types are included for clarity):

1. Compile the foreign function (SQUARE.PAS) to create its object file; the object file will be located in the current default directory (not the current program library):

   ```
   $ PASCAL SQUARE.PAS
   ```

2. Compile the associated Ada specification (SQR_.ADA) and the calling subprogram (ADA_CALLER.ADA); the resulting object files will be located in the current program library (not the current default directory). Note that compiling the specification of a unit that has a foreign body does not cause the body to become obsolete.

   ```
   $ ADA SQR_.ADA, ADA_CALLER.ADA
   ```

3. Copy (or enter) the foreign object file (SQUARE.OBJ) into the current program library as the body of function specification SQR:

   ```
   $ ACS COPY FOREIGN SQUARE.OBJ SQR
   ```

After you execute these commands, you can use the ACS LINK command to link ADA_CALLER and SQR, as follows:

```
$ ACS LINK ADA_CALLER
```

If you have a number of non-Ada routines that need to be called (imported) by an Ada main program, you can simplify the linking operation by writing a package that specifies the imported routines and has an imported linker options file as its body. For example, assume you have the following package specification:

```
package MANY_ROUTINES is
   subtype STRING_TYPE is STRING(1..25);

   function READ_STRING (X: STRING_TYPE) return STRING_TYPE;
   pragma INTERFACE (PASCAL, READ_STRING);

   procedure SORT_STRING (X: STRING_TYPE);
   pragma INTERFACE (PLI, SORT_STRING);

   procedure PRINT_LIST;
   pragma INTERFACE (FORTRAN, PRINT_LIST);
end MANY_ROUTINES;
```

Also assume that you have a linker options file named MANY_ROUTINES_BODY.OPT that contains references to the following .OBJ files:

```
READ_STRING,SORT_STRING,PRINT_LIST
```

After the specification MANY_ROUTINES is compiled, you can enter the linker options file into the current program library as the body of package MANY_ROUTINES, using the ACS ENTER FOREIGN command, as follows:

```
$ ACS ENTER FOREIGN/OPTIONS MANY_ROUTINES_BODY.OPT MANY_ROUTINES
```

Then, assuming that package MANY_ROUTINES is named by a main program in a **with** clause, you can link the main program (and the routines in this package) by entering the ACS LINK command. The linker options file is appended to the command file generated by the ACS LINK command to perform the linking operation.

4.2.2 Using the ACS LINK Command

The ACS LINK command has two forms that allow you to link Ada units directly with foreign files, in cases where you do not want to copy or enter the foreign files into your program library. The first form allows you to link foreign files with a VAX Ada main program:

```
ACS LINK/MAIN VAX-Ada-main-program-name [file-spec[,...]]
```

In VAX Ada, a main program is a procedure or function with no parameters; if it is a function, it must return a value of a discrete type. A main program can also be a procedure declared with the pragma EXPORT_VALUED_PROCEDURE that has one formal **out** parameter that is of a discrete type. The ACS LINK command assumes the /MAIN qualifier by default.

The second form allows you to specify that the image transfer address is in one of the foreign files (a foreign file is the main program):

```
ACS LINK/NOMAIN unit-name[,...] file-spec[,...]
```

With this form, one or more VAX Ada units may be specified and may be listed in arbitrary order. At least one foreign file containing the image transfer address must also be specified. The file containing the image transfer address must be specified according to the requirements of the particular language.

With either form of the ACS LINK command, you can specify the following kinds of VMS (foreign) files:

- Object files—By default, the ACS LINK command assumes that the specified file is an object file, with a default file type of .OBJ.
- Object libraries or shareable image libraries—When specifying an object library file or a shareable image library file, you must append the /LIBRARY qualifier to the file specification. The default file type is .OLB.

You can also append the /INCLUDE qualifier to an object library file or shareable image library file specification to link particular library modules against your VAX Ada units. If you use the /INCLUDE qualifier, you do not also have to use the /LIBRARY qualifier. The default file type for the library file specification is .OLB.

- Linker options files—When specifying a linker options file, you must append the /OPTIONS qualifier to the file specification. The default file type is .OPT.
- Shareable image files—When specifying a shareable image file, you must append the /SHAREABLE qualifier to the file specification. The default file type is .EXE.

You can use the /USERLIBRARY qualifier to tell the linker to also search user-defined default libraries after it has searched any specified libraries.

By default, VAX Ada units are linked against the default system libraries: the linker first searches the system default shareable image library (SYS$LIBRARY:IMAGELIB.OLB) and then the system default object library (SYS$LIBRARY:STARLET.OLB) to resolve references to routines and symbols not defined in the specified units or files. If you specify the /NOSYSLIB command qualifier, neither of these libraries is searched. If you specify the /NOSYSSHR command qualifier, only SYS$LIBRARY:STARLET.OLB is searched.

The following examples show the use of the ACS LINK command with foreign files. In the first example, the linker is instructed to link the main program HOTEL against the user library NETWORK.OLB and to use the linker options file NET.OPT:

```
$ ACS LINK HOTEL NETWORK.OLB/LIBRARY,NET.OPT/OPTIONS
```

In the next example, the linker is instructed to link two Ada units (FLUID_VOLUME and COUNTER) with a foreign main program (MONITOR.OBJ):

```
$ ACS LINK/NOMAIN FLUID_VOLUME,COUNTER MONITOR.OBJ
```

4.2.3 Using the ACS EXPORT and DCL LINK Commands

The ACS EXPORT command allows you to export Ada object files from your current program library to another directory, so that you can subsequently link them with foreign programs using the DCL LINK command.

The ACS EXPORT command creates an object file that contains the code for all units in a closure of VAX Ada units. The file also contains code to elaborate any library packages in the closure.

By default, the exported object file does not include an image transfer address (in other words, the ACS EXPORT command assumes the /NOMAIN qualifier by default). To include an image transfer address and thus identify an exported Ada unit as a main program, use the /MAIN qualifier with the EXPORT command. The image transfer address applies to the first Ada unit specified with the command.

The object file created with the ACS EXPORT command has the following default file specification:

```
SYS$DISK:[]first-unit-name.OBJ
```

SYS$DISK is a system and/or process logical name that generally represents your default disk, and [] represents your current default directory, not your program library.

You can use the /OBJECT=file-spec qualifier to provide another file specification for the object file.

Any exported units that are to be called from a foreign module must contain the appropriate export pragma in the source code: EXPORT_FUNCTION, EXPORT_PROCEDURE, EXPORT_VALUED_PROCEDURE, EXPORT_OBJECT, PSECT_OBJECT, or EXPORT_EXCEPTION. For example, to export the Ada procedure SWAP, you must include the pragma EXPORT_PROCEDURE (see the *VAX Ada Language Reference Manual* and *VAX Ada Run-Time Reference Manual* for exact details):

```
procedure SWAP (A,B: in out INTEGER) is
    . . .
begin
    . . .
end;
pragma EXPORT_PROCEDURE (SWAP);
```

The following examples show the use of the ACS EXPORT command. In the first example, the EXPORT command creates the object file QUEUE.OBJ. The file contains the code for all units in the closure of QUEUE and QUEUE_MANAGER, including any package elaboration code. The file does not contain an image transfer address.

```
$ ACS EXPORT QUEUE, QUEUE_MANAGER
```

Note that object files created by different invocations of the ACS EXPORT command may include some code that is common—for example, if each closure includes the predefined unit TEXT_IO. In such cases, you cannot link those files into the same image. Whenever the closures could include units in common, you should specify all the units in a single EXPORT command line, as in the preceding example.

The next example creates the object file EXP_HOTEL.OBJ that contains the code for all units in the closure of HOTEL, including any package elaboration code and the image transfer address:

```
$ ACS EXPORT/MAIN HOTEL/OBJECT=EXP_HOTEL
```

The ACS EXPORT command is affected by and can affect the value of SYSTEM.SYSTEM_NAME. In particular, the /SYSTEM_NAME qualifier to this command allows you to target the resulting concatenated object file to a particular value of SYSTEM.SYSTEM_NAME. See Chapter 5 and Appendix A for more information.

4.3 Processing and Output Options

The ACS LINK command has a number of qualifiers that allow you to control how the link operation is processed and what kind of output you will receive. For example:

- You can use the /WAIT or /SUBMIT qualifiers to control whether the link operation is executed in a subprocess or as a batch job.
- You can use the /COMMAND qualifier to save the linker DCL command file (which invokes the linker) and the package-elaboration object file generated by the program library manager.
- You can use the /[NO]MAP qualifier to create a linker map file. When using the /[NO]MAP qualifier, you can specify the /BRIEF, /FULL, and /[NO]CROSS_REFERENCE qualifiers to vary the type and amount of information.
- You can use the /OUTPUT=file-spec qualifier to direct ACS output to a file. The options for directing ACS and linker messages to the terminal or to an output file with the ACS LINK command are the same as those for directing compiler messages with the ACS COMPILE and RECOMPILE commands (see Chapter 3).
- You can use the /[NO]DEBUG and /[NO]TRACEBACK qualifiers to control the presence of debug symbol records and traceback information in the executable image.

You cannot create a shareable image with the ACS LINK command.

The following sections discuss some of these options. For detailed information on all of them, see Appendix A. For more information on the linker map file, see the *VMS Linker Utility Manual*; for more information on the /[NO]DEBUG qualifier, see Chapter 6.

4.3.1 Conventions for Defaults, Symbols, and Logical Names

When the program library manager executes the ACS LINK command, it uses the command file it creates to transmit the current definitions of certain defaults, symbols, and logical names to the processing environment (batch or subprocess). Specifically:

- It preserves the current default directory. Then, by default, any new files are created in that directory.
- It transmits the current definition of the symbol LINK. For example, consider the following symbol definition:

  ```
  $ LINK == "LINK/DEBUG"
  ```

  Then, the following commands have the same effect:

  ```
  $ ACS LINK/MAP HOTEL
  $ ACS LINK/DEBUG/MAP HOTEL
  ```

4.3.2 Executing the Link Operation in a Subprocess or in Batch Mode

By default, the link operation for the ACS LINK command is executed in a subprocess. (The default mode is equivalent to specifying the /WAIT qualifier with the ACS LINK command.) The program library manager creates a spawned subprocess and invokes the DCL command file that invokes the linker. Your current process is suspended while the program library manager executes the command, and you must wait until the command terminates before you can enter another command. The net effect is like executing the command interactively.

By specifying the /SUBMIT qualifier, you can execute the link operation for the ACS LINK command in batch mode. In the following example, the program library manager submits the linker command file for the program HOTEL as a batch job:

```
$ ACS LINK/SUBMIT HOTEL
```

All batch options available with the ACS COMPILE and RECOMPILE commands are also available with the ACS LINK command (see Chapter 3).

4.3.3 Saving the Linker Command File and Package Elaboration File

When you use the ACS LINK command, the program library manager creates a DCL command file for the linker and an object file that elaborates all library packages in the closure of the units specified. By default, the program library manager deletes both the command file and the object file when the ACS LINK command terminates.

You can use the /COMMAND[=file_spec] qualifier to save the command file and optionally provide a file specification. The default file specification for the command file is as follows:

```
SYS$DISK:[]first-unit-name.COM
```

SYS$DISK is a system and/or process logical name that generally represents your default disk, and [] represents your current default directory, not your program library.

When you use the /COMMAND qualifier, the program library manager does not invoke the linker. You can edit the command file and later submit it as a batch job, using the DCL SUBMIT command. Use of the DCL SUBMIT command allows you to use certain batch qualifiers that are supported by DCL but not by the program library manager.

When you use the /COMMAND qualifier, the program library manager also saves the package-elaboration object file. The default file specification for the object file is as follows:

```
SYS$DISK:[]first-unit-name.OBJ
```

You can use the /OBJECT=file-spec qualifier to choose an alternative file specification.

Chapter 5

Managing Program Development

Ada program development often involves more than creating, compiling, linking, executing, and debugging Ada programs. In particular, large projects, involving many programmers and large numbers of Ada compilation units, need to be managed efficiently.

This chapter addresses some of the problems involved in managing program development, and presents information that you can use to solve those problems when working with VAX Ada.

5.1 Decomposing Your Program for Efficient Development

Efficient development involves saving compilation and recompilation time. Separate compilation is a feature of the Ada language that allows you to decompose your application into parts, so that you can compile and recompile the parts that change frequently without having to compile and recompile the entire application.

As discussed in Chapter 1, the following parts, or compilation units, of an Ada program can be compiled separately:

- Package specifications and bodies
- Subprogram specifications and bodies
- Generic unit (subprogram and package) specifications and bodies
- Generic instantiations (subprogram and package) of generic units
- Subunits

An efficiently decomposed program consists of three groups of compilation units:

- The specifications of each functionally coherent part of the program. A functionally coherent part comprises one or more operations (and any related type definitions, object declarations, and so on) needed to perform a certain task or group of related tasks. For example, the package SCREEN_IO is a functionally coherent part of the hotel reservation program because it defines the operations needed to perform the task of screen input-output; a general package of all possible input-output operations would not be a functionally coherent part.
- The bodies that implement the specifications.
- Subunits that further decompose the bodies. Each subunit may itself be divided into smaller subunits.

In general, changes occur most often in the compilation units comprising the implementation, rather than in the specifications. Because this decomposition method suggests concentrating the implementation in subunits, and subunits usually do not have dependent units, you can change and recompile the units that implement each functionally coherent part of the program without having to recompile most or all of the rest of the program. (A compilation unit depends on a body or subunit only when a pragma INLINE or INLINE_GENERIC is involved.)

By using generic units to consolidate common kinds of packages and subprograms across different areas of your implementation, you can also save development, compilation, and recompilation time.

NOTE

Use the ACS LOAD, COMPILE, and RECOMPILE commands to efficiently compile and recompile units without having to determine the order of compilation or which units have become obsolete. See Chapter 3 and Appendix A for more information on these commands.

You can reduce the compilation load on your system by putting each compilation unit (specification, body, subunit, and so on) into a separate source file. Be sure to use the file-name conventions described in Chapter 1.

See the *VAX Ada Language Reference Manual* for more information on packages, generic units, and subunits. See Chapter 1 for more information on unit dependences.

Example 5–1 is a simple application that is decomposed into a main program and a generic package. The package is further decomposed into a specification, body, and subunits.

Example 5–1: Decomposed Stack Application

```
generic

   type ELEMENT_TYPE is private;
   SIZE: INTEGER := 3;

package STACKS is

   type STACK_TYPE is array (INTEGER range <>) of ELEMENT_TYPE;
   type STACK is
      record
         TOP: INTEGER;
         ELEMENTS: STACK_TYPE(1..SIZE);
      end record;

    -- CREATE sets up a new stack.
    --
    procedure CREATE (X: in out STACK);

    -- PUSH adds ELEMENT to the stack, sets OK to TRUE
    -- if successful and to FALSE otherwise.
    --
    procedure PUSH (X: in out STACK;
                    ELEMENT: in ELEMENT_TYPE;
                    OK: out BOOLEAN);

    -- POP sets ELEMENT to whatever is popped, sets OK to TRUE
    -- if successful and to FALSE otherwise.
    --
    procedure POP (X: in out STACK;
                   ELEMENT: out ELEMENT_TYPE;
                   OK: out BOOLEAN);

    -- EMPTY returns TRUE if the stack is empty and FALSE otherwise.
    --
    function EMPTY (X: in STACK) return BOOLEAN;

    -- FULL returns TRUE if the stack is full and FALSE otherwise.
    --
    function FULL (X: in STACK) return BOOLEAN;

end STACKS;
```

(continued on next page)

Example 5–1 (Cont.): Decomposed Stack Application

```
------------------------------------------------------------

package body STACKS is

    procedure CREATE (X: in out STACK) is separate;

    procedure PUSH (X: in out STACK;
                    ELEMENT: in ELEMENT_TYPE;
                    OK: out BOOLEAN) is separate;

    procedure POP (X: in out STACK;
                   ELEMENT: out ELEMENT_TYPE;
                   OK: out BOOLEAN) is separate;

    function EMPTY (X: in STACK) return BOOLEAN is separate;

    function FULL (X: in STACK) return BOOLEAN is separate;

end STACKS;

------------------------------------------------------------

separate (STACKS)
procedure CREATE (X: in out STACK) is
begin
    . . .
end CREATE;

------------------------------------------------------------

separate (STACKS)
procedure PUSH (X: in out STACK;
                ELEMENT: in ELEMENT_TYPE;
                OK: out BOOLEAN) is
begin
    . . .
end PUSH;

------------------------------------------------------------

separate (STACKS)
procedure POP (X: in out STACK;
               ELEMENT: out ELEMENT_TYPE;
               OK: out BOOLEAN) is
begin
    . . .
end POP;
```

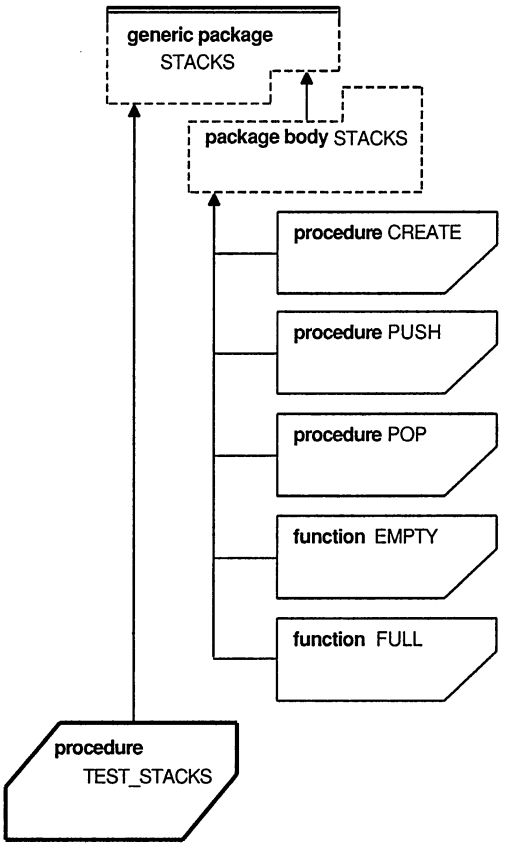
(continued on next page)

Example 5–1 (Cont.): Decomposed Stack Application

```
------------------------------------------------------------

separate (STACKS)
function EMPTY (X: in STACK) return BOOLEAN is
begin
    . . .
end EMPTY;

------------------------------------------------------------

separate (STACKS)
function FULL (X: in STACK) return BOOLEAN is
begin
    . . .
end FULL;

------------------------------------------------------------

with TEXT_IO; use TEXT_IO;
with STACKS;
procedure TEST_STACKS is

   -- Main program that instantiates and uses the stack operations.
   --
   subtype STRING_TYPE is STRING(1..5);

   package INTEGER_STACK is new STACKS(INTEGER,3);
   use INTEGER_STACK;

   package STRING_STACK is new STACKS(STRING_TYPE,3);
   use STRING_STACK;
    . . .

begin
   -- Do some work with the stacks and stack operations.
    . . .
end TEST_STACKS;
```

Figure 5–1 diagrams the application in Example 5–1 to show the unit dependences. Note that because the procedure TEST_STACKS instantiates the generic package STACKS, the procedure itself is still current (unless an inline pragma or equivalent applies), but the instantiations must be completed if the package body or subunits of the package STACKS are compiled again or recompiled. See Chapter 1 and Chapter 3 for more information on incomplete units, obsolete units, and generic completions.

Figure 5–1: Diagram of Decomposed Stack Application



5.2 Setting up an Efficient Program Library Structure

Ideally, you should consider the following factors when setting up a program library and sublibrary structure:

- The structure of the application
- The number of programmers developing the application
- Whether or not the application is going to be run on more than one target

- Whether or not all of the software is being written from scratch
- Whether you will need to produce different versions of the application as it changes over time (for example, Versions 1.0, 1.1, and 1.2)

Figure 5–2 shows a library structure for the decomposed stack application from Example 5–1. Note the following points about Figure 5–2:

- The top-level program library contains the generic package specification STACKS.
- The immediate sublibraries contain the body of STACKS and the main program TEST_STACKS; two sublibraries are used because this application is being developed for two targets: VMS and VAXELN. Off-the-shelf or other prewritten source code could also go in these sublibraries.
- Programmers work in lower-level sublibraries to develop the subunits of STACKS. Although four sublibraries are shown, any number of sublibraries could be used to develop STACKS and its subunits.

Figure 5–2 does not show multiple versions of the application, but additional sublibraries could be used to create and develop different versions or other development streams.

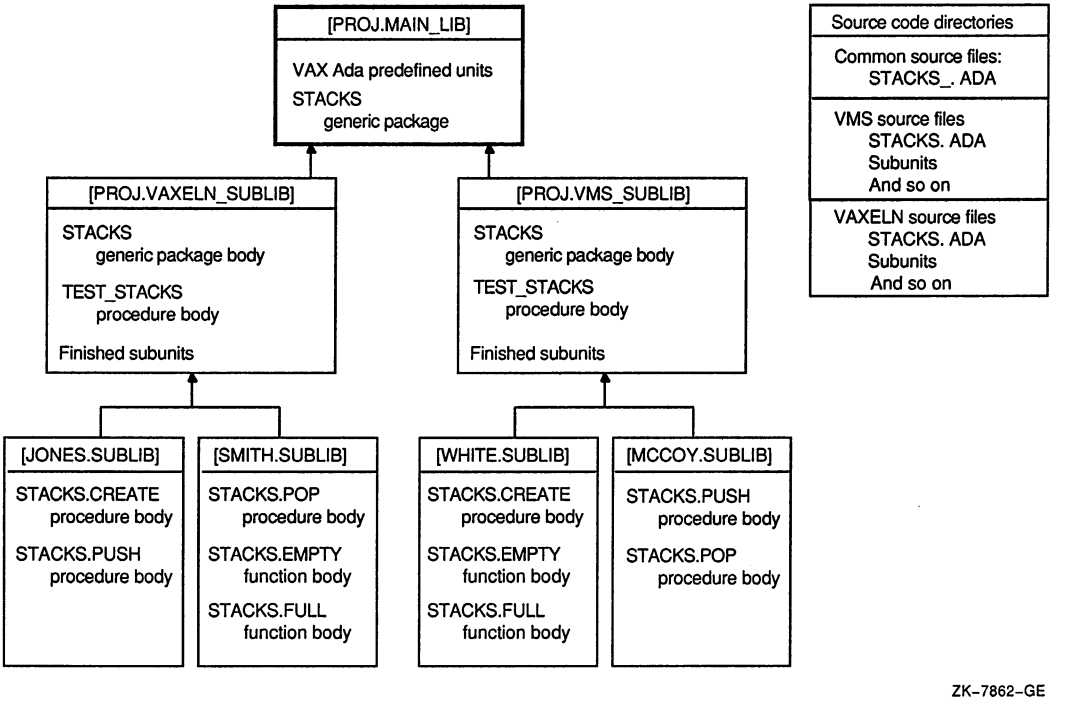
A structure like the one in Figure 5–2 allows testing from the bottom up. See Chapter 2 for additional information on developing and testing units in sublibraries.

After programmers have developed and tested new, stable versions of the units in the application, they return the units to the appropriate project source code directory. For simplicity, Figure 5–2 shows one source code directory to the right of the program library structure. See Section 5.3.1 for more information on setting up and managing source code directories.

You can make the new, stable units available to the other programmers in a number of ways:

- You can merge or copy units from the sublibraries into the more global parent libraries.
- You can compile the units from the source code directory into the appropriate parent libraries.

Figure 5–2: Efficient Program Library and Sublibrary Structure



When you use the ACS MERGE command, be careful to enter it at the right level. For example, if you use a low-level sublibrary to modify and test a new specification, and you merge the specification into its immediate parent library, the specification may end up in the sublibrary containing the package bodies rather than in the library containing the specifications. In this case, you may want to do one of the following operations:

- Copy (rather than merge) the new specification to its appropriate location
- Create a temporary sublibrary at the correct level, copy the specification to that sublibrary, and merge from there

- Change the parent of the sublibrary you are working in before doing the merge

See Chapter 2 for more information on merging units and changing the parent of a sublibrary.

Merging or copying units from the sublibraries to the more global parent libraries has the advantage that the new units are immediately available to other programmers on the project. However, the replacement of these units may cause other units in upper- as well as lower-level libraries to become obsolete. The obsolete units must then be recompiled to become current again. If recompilations are required too often, they may disrupt the work being done by individual programmers on the project. Also, the source code directories must be carefully maintained in parallel with the program libraries.

To minimize the impact of the replacements, you can update upper-level libraries by compiling the new source files from the project source directories at known times using the ACS LOAD and ACS COMPILE or RECOMPILE commands. Again, individual project members may need to recompile obsolete units in their sublibraries. However because the updating of parent libraries is done at known times, the impact on project members is controlled and less disruptive. An advantage of this method is that maintenance of the source code directories is synchronized with management of the program libraries. A disadvantage of this method is that new units are not immediately available to all members of the project.

Depending on the scope and complexity of your application, you may need to protect your library structure from regressions caused by updates. To achieve this protection, you can set up a separate library structure that parallels your upper-level working libraries. Then, you can build the complete application and perform regression tests on it in the separate library structure. After the tests are successful, you update the working libraries as previously discussed:

- By copying the units from the separate libraries into your upper-level working libraries, while also updating the source code directories.
- By updating the source code directories first, and then compiling the units from the source code directories into the working libraries.

5.3 Integration with Other VAX Tools

Like other VAX languages and layered products, VAX Ada is designed to be used with a variety of Digital software development tools (see Chapter 1). This section discusses how you can use the following tools with VAX Ada to manage program development:

- VAX Language-Sensitive Editor (LSE)
- VAX DEC/Code Management System (CMS)

General-purpose (as opposed to management) development tools are discussed elsewhere in this manual:

- For general and Ada-specific information on using LSE and the VAX Source Code Analyzer (SCA), see Appendix C.
- For general and Ada-specific information on using the VMS Debugger, see Chapter 6.

For general information on creating a software environment, see *A Methodology for Software Development Using VMS Tools*. This manual describes how to create a development environment using the VAX Software Engineering Tools (VAXset). VAXset includes LSE, SCA, CMS, as well as the VAX DEC/Module Management System (MMS), VAX DEC/Test Manager, and VAX Performance and Coverage Analyzer (PCA).

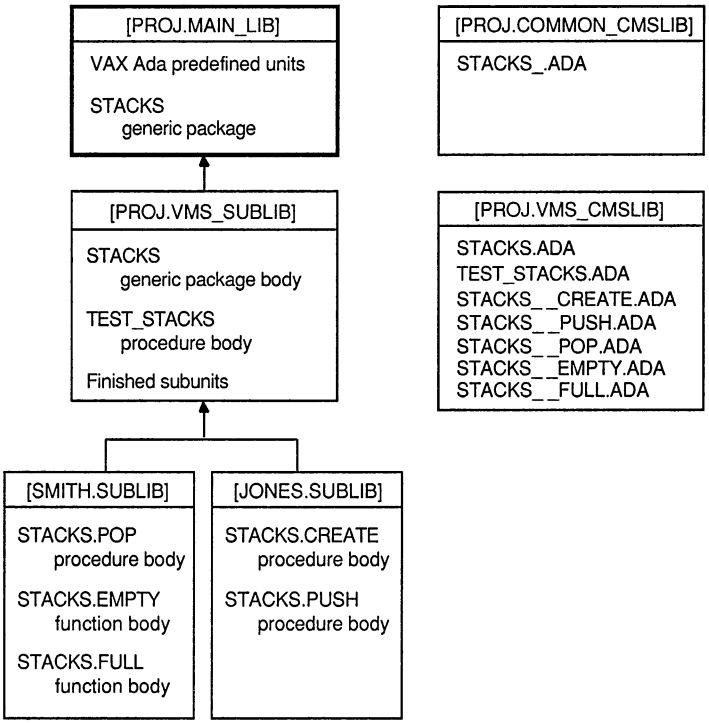
5.3.1 Setting up Source Code Directories

An effective way to set up and manage source code directories is to use CMS. The *Guide to VAX DEC/Code Management System* and *A Methodology for Software Development Using VMS Tools* give detailed information on how to use CMS.

You can use CMS libraries in conjunction with VAX Ada program libraries and sublibraries. You can have a single CMS library for all of your source code, and use that library in conjunction with a number of VAX Ada program libraries. Or, you can divide up your source code among several CMS libraries that are associated with one or more VAX Ada program libraries.

Beginning with Version 3.0, CMS allows you to use search lists to manage multiple libraries. So, you can construct trees of CMS libraries that parallel your VAX Ada program libraries and sublibraries. Figure 5–3 shows one such configuration.

Figure 5–3: Ada Program Library and Sublibrary Structure with CMS Libraries



The following search list applies to the library structure in Figure 5–3:

```
$ CMS SET LIBRARY [PROJ.VMS_CMSLIB], [PROJ.COMMON_CMSLIB]
```

When searching for library elements, CMS starts with the first library on the list and stops when it finds the first unit that meets whatever requirements you have specified (RESERVE element-name, FETCH/GENERATION=2 element-name, and so on). Thus, a search list like the one in this example causes source code modules in a lower-level CMS library to hide source code modules with the same name higher-level libraries. This effect is similar to the panes-of-glass effect you get when you use VAX Ada sublibraries for compilations, and you can use it for retrieving and modifying source code in the same way that you use VAX Ada sublibraries to test Ada compilations (see Chapter 2).

5.3.2 Managing Source Code Modifications

LSE, CMS, and the VAX Ada program library manager offer a number of features that allow you to manage source code modifications. For example, LSE allows you to retrieve Ada source code from a CMS library, modify it, and then compile it from within the editor (see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer* for more information on how LSE is integrated with CMS).

Once you have moved an Ada source code element from a CMS library into an LSE editing buffer, you can use the LSE COMPILE/REVIEW command to compile it into your current Ada program library or sublibrary. The LSE COMPILE/REVIEW command causes the compilation to take place in a subprocess.

Note that when you use the /REVIEW qualifier for this operation, your process will wait until the subprocess completes. By not using the /REVIEW qualifier, you can keep working in the editor, and later use the LSE REVIEW command to read the diagnostics files after the compilation completes.

An alternative method of compiling from within LSE is to use a command procedure that causes the compilation to take place in a batch queue. For example, the command procedure in Example 5–2 sends all Ada compilations to whatever queue is represented by the logical name ADA$BATCH.

Example 5–2: Command Procedure for Doing LSE Ada Compilations in Batch Mode

```
$! Command procedure for compiling Ada source code in an LSE buffer
$! using the ADA$BATCH queue. For this command procedure to succeed,
$! you must have a current program library (use the ACS SET LIBRARY
$! command), and you must have defined the logical name ADA$BATCH.
$!
$! Parameters passed by LSE:
$!   P1 = Source file specification
$!   P2 = Additional qualifiers (/DIAGNOSTICS, for example)
$!
$ NAME = F$PARSE(P1,,,"NAME")
$ SET NOON
$ DELETE 'NAME'.COM;*
$ PURGE ''NAME'.LOG"
$ PURGE/NOLOG 'NAME'.DIA
$ SET ON
```

(continued on next page)

Example 5–2 (Cont.): Command Procedure for Doing LSE Ada Compilations in Batch Mode

```
$ OPEN/WRITE COMFILE 'NAME'.COM
$ IF F$TYPE(ada) .EQS. "" THEN ada = "ada"
$ DEFDIR = F$TRNLNM("SYS$DISK")+F$DIR()
$ WRITE COMFILE "$ SET DEFAULT ''DEFDIR'"
$ WRITE COMFILE "$ ''ADA' /LIBRARY='F$TRNLNM("ADA$LIB") ''P1' ''P2'
$ CLOSE COMFILE
$ SUBMIT/NOPRINT/QUEUE=ADA$BATCH/LOG_FILE='DEFDIR''NAME'.LOG -
  'NAME'.COM
```

To use this command procedure from within LSE, enter (or define a key for) the LSE COMPILE command, giving the batch-job command procedure as an argument. For example:

```
LSE Command> COMPILE @ADA_BATCH.COM
```

Alternatively, you can compile your Ada source code outside of the editor (preferably as a batch job), append all of the diagnostics files, and review them all at once during an editing session.

The LSE COMPILE command uses the DCL ADA command to perform its Ada compilations, which makes it useful for compiling single units, but not for compiling or recompiling a set of units (execution closure).

An alternative to compiling using LSE is to compile using the ACS LOAD or COMPILE commands. In both cases, you can obtain diagnostics files for review within LSE by using the /DIAGNOSTICS qualifier (see Appendix A for more information on the behavior of this qualifier with these commands).

You can also use the ACS LOAD or COMPILE commands to compile Ada units from a CMS library.

In the following example, the first command sets up a search list of CMS libraries. The second command fetches from those libraries the generations of Ada source code elements that are associated with the class BASELEVEL_4. The third command loads the Ada source code elements into the current Ada program library.

```
$ CMS SET LIBRARY DISK:[PROJ.CMSUBLIB1],[PROJ.CMSLIB]
$ CMS FETCH *.ADA/GENERATION=BASELEVEL_4
$ ACS LOAD *.ADA
```

Once a set of Ada units exists in your current program library, you can use the ACS SET SOURCE and COMPILE commands to cause the latest generation of modified units existing in a CMS library to be compiled again. In the following example, the first command sets the CMS library. The second command establishes a source file search list for the ACS COMPILE command. The third command causes the closure of the unit TEST_STACKS to be compiled from the source files stored in the CMS library denoted by CMS$LIB.

```
$ CMS SET LIBRARY DISK:[PROJ.CMSUBLIB1],[PROJ.CMSLIB]
$ ACS SET SOURCE CMS$LIB
$ ACS COMPILE/LOG/NODATE_CHECK/CLOSURE TEST_STACKS
```

If the execution closure of the units has changed since the external source files were last compiled, then use the /PRELOAD qualifier with the ACS COMPILE command. If you have added new units to the execution closure, you must put them in the library before entering the ACS COMPILE command. You can put new units in the current program library either by compiling them with the DCL ADA command or by loading them with the ACS LOAD command,

If you need to work with a different generation, class, or group for your ACS LOAD or COMPILE compilation, use the following procedure:

1. Use CMS to fetch or reserve the generation, class, or group you want to compile from.
2. Put the resulting Ada files in a temporary VMS directory.
3. If you are using the ACS COMPILE command, use the ACS SET SOURCE command to include the temporary directory in the search list for the ACS COMPILE command. If you are using the ACS LOAD command, give the temporary VMS directory specification directly on the command line.
4. If you are compiling the Ada files for the first time, enter the ACS LOAD command and then enter the ACS COMPILE command. If you are updating a library with modified units, enter only the ACS COMPILE command.
5. Clean up the temporary directory.

5.4 System Considerations

When configuring a system to handle Ada compilations, you need to consider all of the activities that will be performed. For example, you need to consider the amount of parallel compilation that the system can take, as well as considering the amount of memory you will need.

In general, you should allow 2 megabytes of physical memory for each concurrent Ada compilation. Given the 2-megabyte requirement, you can allow as many concurrent compilations as you have VAX units of performance (VUPs). One VUP is equal to the performance of a VAX-11/780. Appendix E gives detailed information on the memory requirements for VAX Ada, as well as information on tailoring the use of system resources.

In addition, consider the following suggestions:

- Plan to minimize the size of your compilations by decomposing your applications and structuring your libraries and sublibraries as shown in Section 5.1.
- Use a dedicated batch queue to serialize compilations. By using a batch queue, you can have a large working set size (as well as other parameters) for the batch queue, while minimizing the working set size of each individual account on the system. See Appendix E for additional information.
- Use VAXclusters if at all possible. VAX Ada operations across DECnet can be slow because process and file access links can accumulate (a consequence of doing any operation across DECnet, not just VAX Ada operations). See Section 5.5 for more information on configuring and using VAX Ada program libraries across DECnet. See the *VMS VAXcluster Manual* for more information on configuring and managing VAXclusters.

5.5 Distributed Programming Considerations

Program libraries can be accessed across DECnet, making possible distributed program development among program libraries that are not all local to a single VMS system. For example, you can have a sublibrary on a different node from its parent library, and you can enter or copy units from program libraries that reside on different nodes. This ability allows you to do compilation on a VAXstation, while still using program libraries on another system.

The following sections give some guidelines for specifying program libraries across DECnet. In particular, Section 5.5.5 lists any restrictions that may apply.

5.5.1 Configuring a Library Structure Across DECnet

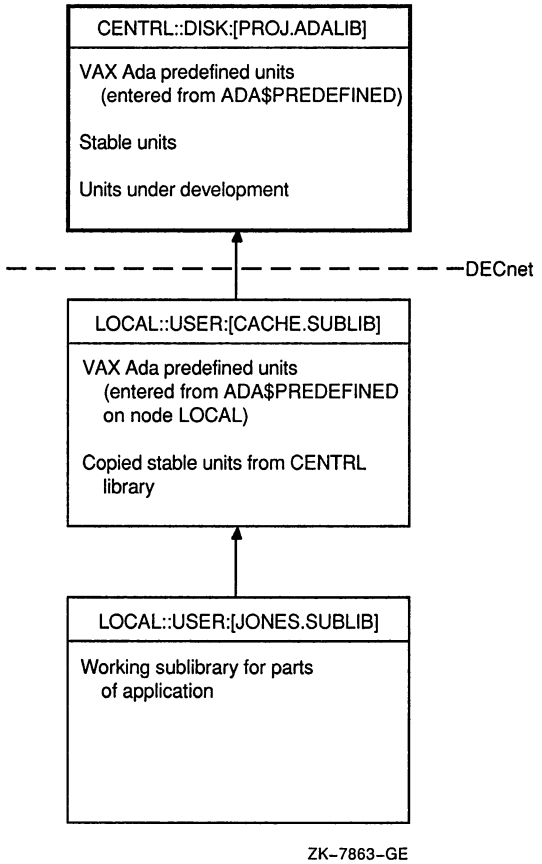
If you plan to configure a system across DECnet, be careful about causing access links to accumulate. Instead, consider caching those units that are constant or finished (such as the units in ADA$PREDEFINED) in a local library to minimize access across DECnet. Figure 5–4 suggests one such configuration:

- The central program library—DISK:[PROJ.ADALIB]—is on node CENTRL. This library contains the units entered from the ADA$PREDEFINED library on that node, some finished units, and some units under development.
- A cache sublibrary—USER:[CACHE.SUBLIB]—is on node LOCAL. The central program library is the parent of this sublibrary. This cache sublibrary contains units from the ADA$PREDEFINED library on node LOCAL and any finished units copied from the central program library. Units that are too large to copy over the network or that need to be monitored as they change remain in the central program library.
- A working sublibrary—USER:[JONES.SUBLIB]—is also on node LOCAL. The cache sublibrary is the parent of this sublibrary. This sublibrary contains units that are being developed by a programmer on node LOCAL.

Note that in this situation, to make the local sublibrary's units available to all of the users of this system, you must merge twice: once from the working sublibrary to the cache sublibrary, and once from the cache sublibrary to the central library.

There are other library caching schemes that may be more appropriate to your application. For example, you might set up a cached library on the local node that is a snapshot of an independent library on another node. You can then enter units from the cached library into your working library. Note that you cannot recompile entered units.

Figure 5–4: DECnet Program Library Configuration



When working with libraries across DECnet, be sure to consider system security. For maximum security, use proxy accounts (see the *Guide to DECnet-VAX Networking* and the *VMS Networking Manual* for more information). For example, the node CENTRL would allow proxy access to the user JONES from node LOCAL. Proxy access also improves performance because it causes the system to reuse access links, which can otherwise accumulate as you perform parent library operations across DECnet.

5.5.2 Accessing a Program Library Across DECnet

When you access a program library on a single VMS system, you must be aware of user-identification-code (UIC) protection mechanisms such as UIC-based system, user, or group protection (see Section 5.6). When you access a program library across DECnet, you must also be aware of UIC-based world protection, as well as protocols such as proxy accounts.

There are three ways to open, read, and write a program library or sublibrary on another node:

- You can give the files world read protection. Read protection is sufficient if only read access is desired and it does not matter that every user in the network can read the files.
- You can create proxy accounts. Proxy accounts are useful if both read and write access is desired.
- You can change the protection of all files accessed across the network to world read and write. However, for security reasons, this is usually not a good idea.

All operations that involve accessing a unit across the network must have read access from the node accessing the program library to the node specified in the files associated with the unit. Proxy accounts provide one way of making this access available in a more controlled manner. Note that write access is required for the parent of a sublibrary when you use the ACS MERGE command. See Section 5.6.1 for information on the kinds of program library access required for the various ACS commands.

For example:

```
ACS> SET LIBRARY USER:[TEST.ADALIB]
%I, Current program library is USER:[TEST.ADALIB]
ACS> ENTER UNIT CENTRL::DISK:[PROJ.ADALIB] STACKS,TEST_STACKS/LOG
%I, STACKS entered
%I, TEST_STACKS entered
ACS> DIR STACKS_,TEST_STACKS/FULL
STACKS
   generic package             16-Aug-1988 14:33
     @ CENTRL::DISK:[PROJ.ADALIB]STACKS.ACU;1
     @ CENTRL::DISK:[PROJ.ADALIB]STACKS.ADC;1
     @ CENTRL::DISK:[PROJ]STACKS.ADA;2

TEST_STACKS
   procedure body              16-Aug-1988 14:33
     @ CENTRL::DISK:[PROJ.ADALIB]TEST_STACKS.ACU;1
     @ CENTRL::DISK:[PROJ.ADALIB]TEST_STACKS.ADC;1
     @ CENTRL::DISK:[PROJ]TEST_STACKS.ADA;3
```

After STACKS and TEST_STACKS have been entered from the library CENTRL::DISK:[PROJ.ADALIB] into the library USER:[TEST.ADALIB], any operations that access STACKS and TEST_STACKS in USER:[TEST.ADALIB] must have read access to the program library files for those units.

See the *Guide to Maintaining a VMS System* for more information on how to create proxy accounts.

5.5.3 Achieving Efficient DECnet Access to Program Libraries

Pay careful attention to SYSGEN (System Generation Utility) and DECnet parameters (on both the local and remote nodes) that may affect the availability of compilation units or files accessed across DECnet. Your system manager can help you with this.

For example, every time a file is opened on a remote node, a temporary connection, called a *logical link*, is made from the local node to the remote node. The total number of logical links allowed at one time is controlled by DECnet and may be set by the Network Control Program (NCP) Utility with the following command:

```
NCP> SET EXECUTOR MAX LINKS N
```

The N in this command is the maximum number of logical links. This number represents the system (not process) quota; each connection between two nodes deducts one from the quota total. When setting this value, note that both the Ada compiler and the program library manager use the per-process FILLM (file and logical link limit) quota, not the system quota, to limit the total number of open files at one time. Thus, limiting the total number of files open at one time will also reduce the potential number of logical links created to the remote node: a logical link is only required for files that are not accessed locally.

The creation of a logical link may also involve the creation of a process on the remote node. Thus, you may need to increase (or at least monitor) the values of the SYSGEN parameters MAXPROCESSCNT and BALSETCNT to allow more processes to be created for network servers (FALs).

If you are expecting to access a number of files or compilation units across DECnet, you may want to increase the value of the UAF buffered input-output byte count limit (BYTLM) parameter on your system. The value of this parameter affects the efficiency of program library operations performed across DECnet. Other parameters may also have an effect. DECnet parameters are documented in the *Guide to DECnet-VAX Networking*.

After you have set or reset system parameters to accommodate the use of remote nodes, you may want to run the Digital-supplied AUTOGEN command procedure (SYS$UPDATE:AUTOGEN.COM) to recompute optimal values for related parameters. See the *Guide to Maintaining a VMS System* for more information on SYSGEN and AUTOGEN.

5.5.4 Effect of Network Failures

A network failure during a compilation can have several effects. If the failure occurs while the compiler is in operation, the compilation can terminate, leaving your program library in whatever state it was in before the beginning of the compilation. If the failure occurs during a phase in which the program library is being updated, your program library may be in an inconsistent state. You would then have to repair any inconsistencies using the ACS VERIFY command (see Section 5.7.5 and Appendix A).

In other words, a network failure during a compilation is like a system failure during a compilation, except that the network failure does not stop your process from running, and you could receive numerous file-access and Ada diagnostic messages as a result.

5.5.5 Restrictions on Using Program Libraries Across DECnet

Observe the following restrictions when distributing program libraries across DECnet:

- In the absence of the VAX Distributed File System (DFS), CMS (Version 3.0 and lower) does not support access across DECnet. Thus, the program library manager may issue an error if an operation requires accessing a CMS library across DECnet.
- VMS directories cannot be created across DECnet. Thus, the ACS CREATE LIBRARY and CREATE SUBLIBRARY commands can be used to create program libraries or sublibraries across DECnet only if the corresponding VMS directories already exist for those libraries on the remote node.
- Exclusive access to a compilation library on another node is not permitted and results in an error. Therefore, ACS SET LIBRARY/EXCLUSIVE for a program library on a remote node fails with an error.

Because the ACS VERIFY/REPAIR and REORGANIZE commands can depend on the use of ACS SET LIBRARY/EXCLUSIVE, VERIFY/REPAIR and REORGANIZE are also not permitted for a compilation library on another node when they are used in conjunction with SET LIBRARY/EXCLUSIVE.

- A program library on another node must not be opened with an access control string. An error results if such an attempt is made.

5.6 Protecting Program Libraries

The ACS commands require various kinds of access to program libraries. For example, to copy units from a library, you need only read access to the library; but to copy or compile units into a library, you need read and write access to it.

The techniques for controlling access to program libraries are based on those for controlling access to VMS directories. The following topics are discussed in the following sections:

- The kind of library access needed for each ACS command
- The user-identification-code (UIC) based protection for the program library files required for each kind of library access
- The use of VMS access control lists (ACLs) on program libraries for each kind of library access

For complete details on VMS file and directory protection, see the *Guide to VMS System Security*, *VMS DCL Dictionary*, and *VMS Access Control List Editor Manual*.

5.6.1 Program-Library Access Requirements for ACS Commands

The program library manager recognizes three kinds of program library access (not to be confused with VMS UIC-based protection categories):

- Read (R)—means that the library and units in the library can be opened for reading
- Write (W)—means that units in the library can be deleted as well as written
- Delete (D)—means that the library can be deleted (including any units in the library, the library index file, and the VMS directory associated with the library)

Table 5–1 lists the kinds of access required by each of the ACS commands.

Table 5–1: Program Library Access Needed to Use ACS Commands

ACS Command	Library Access	Comments
CHECK	R	
COMPILE	RW	
CONVERT LIBRARY	RW	
COPY FOREIGN	RW	Read access is needed to the directory from which the foreign file is copied.
COPY UNIT	RW	Read access is needed to the program library from which the unit is copied.
CREATE LIBRARY	RW	
CREATE SUBLIBRARY	RW	
DELETE LIBRARY	RWD	
DELETE SUBLIBRARY	RWD	
DELETE UNIT	RW	
DIRECTORY	R	
ENTER FOREIGN	RW	Read access is needed to the directory from which the foreign file is entered.
ENTER UNIT	RW	Read access is needed to the program library from which the unit is entered.
EXPORT	R	
EXTRACT SOURCE	R	
LINK	R	
LOAD	RW	
MERGE	RW	Read-write access is needed to the parent library.
RECOMPILE	RW	
REENTER	RW	Read access is needed to the program library from which the unit is reentered.
REORGANIZE	RW	Exclusive access is also needed.
SET LIBRARY	R	

(continued on next page)

Table 5–1 (Cont.): Program Library Access Needed to Use ACS Commands

ACS Command	Library Access	Comments
SET LIBRARY/EXCLUSIVE	RW	Exclusive access is also needed.
SET LIBRARY/READ_ONLY	R	
SET PRAGMA	RW	
SHOW LIBRARY	R	
SHOW PROGRAM	R	
SHOW VERSION	R	
VERIFY	R	
VERIFY/REPAIR	RW	Exclusive access is also needed.

5.6.2 Standard User-Identification-Code (UIC) Based Program Library Protection

Because they exist in the VMS environment, the files associated with program libraries and the units contained in them inherit a default, standard UIC-based protection when they are created—that is, a protection that is coded for each of four hierarchical protection categories:

System (S)
Owner (O)
Group (G)
World (W)

Each category can be granted any of the following access codes, in any combination:

Read (R)
Write (W)
Execute (E)
Delete (D)

Note that when a UIC delete access code is associated with a file, it means that that individual file can be deleted (as opposed to the program library delete access discussed in Section 5.6.1, which means that an entire program library and its contents can be deleted).

In the context of the VMS environment of directories and files, a program library is a VMS directory that contains a library index file (ADALIB.ALB), a library version control file (ADA$LIB.DAT), and all of the files associated with the compilation units in the library. See Appendix D for a complete description of these files.

When you create a program library or sublibrary by entering an ACS CREATE LIBRARY or CREATE SUBLIBRARY command, the following files are created with the following UIC-based protection:

- The VMS directory associated with the library (if it does not already exist). This directory file inherits whatever protection is in effect for the next-higher-level directory, less any delete access for each unspecified protection category. This inherited protection scheme is consistent with the scheme used by the DCL CREATE/DIRECTORY command.
- The library index file (ADALIB.ALB) and library version control file (ADA$LIB.DAT). These files are created with whatever file protection was most recently specified with the DCL SET PROTECTION/DEFAULT command.

Each time a compilation unit is added to the library, if any files are created in the library (VMS directory) for that unit, those files inherit the same UIC-based protection as the library index file (not the VMS directory file). In addition, if the library index file allows write access for a given protection category, delete access is also given for that category.

Table 5–2 shows how the UIC-based protection for each file in a program library is related to the program library access discussed in Section 5.6.1. Table 5–2 shows the minimum UIC-based protection needed for each kind of program library access. If the minimum UIC-based protection requirements are not met for program library access, then normal library operations may not complete properly. For example, the ACS DELETE UNIT command requires read-write (RW) program library access. Because program library write access also requires UIC delete access, if a file associated with that unit does not allow delete access, the program library manager will not delete the file.

Table 5–2: Minimum UIC Protection for Each Kind of Library Access

Program Library Access (see Section 5.6.1)	Library Index File and Library Version Control File (UIC Access)	Other Library Files (UIC Access)	VMS Directory File (UIC Access)
R	R	R	R
RW	RW	RWD	RW
RWD	RWD	RWD	RWD[1]

[1]If the VMS directory file does not have UIC delete access, it will be left empty (the contents but not the directory file will be deleted).

As shown in Table 5–2, library index file UIC protection must be the same as the VMS directory file protection. To ensure this, you can use the /PROTECTION qualifier when you create the library. However, if the VMS directory file already exists when you enter the ACS CREATE LIBRARY command, its protection will not changed by this qualifier.

For example:

```
$ ACS CREATE LIBRARY -
_$ /PROTECTION=(SYSTEM:RWE, OWNER:RWED, GROUP:R, WORLD) -
_$ [JONES.HOTEL.ADALIB]
```

After this command is executed, the specified protection applies to the directory file [JONES.HOTEL]ADALIB.DIR, the library index file [JONES.HOTEL.ADALIB]ADALIB.ALB, and the library version control file [JONES.HOTEL.ADALIB]ADA$LIB.DAT. Other library files later created in the program library [JONES.HOTEL.ADALIB] will have protections as specified in Table 5–2 for each user category.

Sometimes you need to ensure that a program library is never modified during a program library manager session. You can do this by first invoking the program library manager interactively, and then entering an ACS SET LIBRARY/READ_ONLY command. After you enter this command, any ACS command that requires write or delete library access will fail. For more information about the /READ_ONLY qualifier to the ACS SET LIBRARY command, see Chapter 2 or Appendix A.

5.6.3 Program Library Protection Through Access Control Lists

VMS access control lists (ACLs) offer an alternative method of file protection. You can use this method in conjunction with the standard UIC-based protection described in Section 5.6.2 to tune access control where it is needed.

The central mechanism behind ACLs is a rights database that specifies identifiers and holders of those identifiers, as well as ACLs that relate the identifiers with the access to be granted or denied to the holders of the identifiers. By using ACLs, you can match specific users to the specific access you want to grant or deny.

Each ACL consists of one or more access control list entries (ACEs) that grant or deny access to a particular user or group of users. There are three kinds of ACEs:

- Identifier ACE—Controls the kinds of access to be allowed to a particular user or group of users. An identifier ACE can be a UIC, a general identifier established by the system manager, or a system-defined identifier (for example, BATCH, NETWORK, DIALUP, INTERACTIVE, and so on).
- Default protection ACE—Defines the default protection for a directory, so that the protection can be propagated to the files and subdirectories created in that directory.
- Security alarm ACE—Provides a security alarm when an object is accessed in a particular way.

For a complete description of ACLs see the *Guide to VMS System Security*.

To allow you to tune access to a program library, the program library manager checks for any identifier ACEs on the library index file (ADALIB.ALB) in the VMS directory containing the program library. If there are identifier ACEs defined on the library index file, the program library manager will grant or deny access depending on the kind of program library access required by the ACS operation (see Table 5–1).

As Table 5–2 shows, program library access is always the same as minimum UIC access required for the library index file. Thus, by controlling access to the library index file, you can control access to the program library.

For example, by applying an ACE to the library index file that denies all ACS operations requiring write or delete program library access (such as COMPILE, DELETE UNIT, ENTER UNIT, and so on), you can "freeze" the program library for a particular set of users. The following command restricts all members of the group PROJ to read-only ACS operations:

```
$ SET ACL DISK:[ADALIB]ADALIB.ALB/ACL=(ident=[PROJ,*], access=READ)
```

See the *Guide to VMS System Security* for a complete description of how access requests are evaluated in the presence of ACLs.

Although putting an ACL on the library index file provides the desired access control from the program library manager, it is not sufficient to protect against users using another VMS utility (like DCL) to access the files in the program library. To protect against those users, you need to apply the ACL to all files in the VMS directory associated with the program library, according to the information given in Table 5–2. Normally, you should not need to do this; keep in mind that putting ACLs on all files in the program library will result in performance penalties for both users of the VAX Ada program library manager and the entire VMS system in which those users are working.

Also, do not assume that specifying ACCESS=NONE for an identifier will completely prohibit the holders of the identifier from accessing the library. Users who are in either the SYSTEM or OWNER category are still entitled to whatever access the UIC-based protection affords that category. Furthermore, if the users hold privileges, they will be granted the access requested through the privilege. See the *Guide to VMS System Security* for more information on access request evaluation.

5.7 Maintaining Program Libraries

Program library maintenance involves the following tasks:

- Making references to program libraries independent of specific devices and directories
- Copying program libraries
- Backing up program libraries
- Reorganizing program libraries
- Verifying and repairing program libraries
- Recompiling after new releases of VAX Ada

The following sections discuss these tasks in detail and present information on how to make some of the maintenance activities (copying, backing up, and so on) efficient.

5.7.1 Making References to Program Libraries Independent of Specific Devices and Directories

A program library often references units in other program libraries. A sublibrary, in addition, references its parent library. By making unit references and parent library references device and directory independent, you can enter units, change the parent of a sublibrary, back up, and restore program libraries independent of the device and directory references associated with the units in those libraries. You can also change the parent of a sublibrary (see Chapter 2).

You can achieve device independence by using *concealed-device logical names*. Section 5.7.1.1 discusses concealed-device logical names.

You can achieve device and directory independence, and thus program library reference independence, by using *rooted directory* syntax when specifying parent libraries with the ACS CREATE SUBLIBRARY command or when specifying units in the ACS ENTER command. Section 5.7.1.2 discusses rooted directories.

You can make logical name assignments at the system, group, or job level, as appropriate. The *VAX Ada Installation Guide* instructs your system manager to perform some standard system-wide logical name assignments to public devices.

For more information on concealed-device logical names, rooted directories, and logical names, see the *VMS DCL Dictionary* and the *Guide to VMS File Applications*.

5.7.1.1 Using Concealed-Device Logical Names

A concealed-device logical name has the following properties:

- Its equivalence name contains a physical device name.
- It prevents the equivalence name from being displayed in the file specification that results when the logical name is translated; the logical name is displayed in place of the equivalence name.

To define a logical name as a concealed-device logical name, you must use the /TRANSLATION_ATTRIBUTES=CONCEALED qualifier with the DCL DEFINE or ASSIGN commands. You must also use a physical device name, not a logical device name. For example, the following command assigns the concealed-device logical name DISK to the physical device DBA3:.

```
$ DEFINE/TRANSLATION_ATTRIBUTES=CONCEALED DISK DBA3:
```

After this assignment, the logical name DISK (not the physical device name DBA3:) is displayed in system messages. Also, utilities like the VAX Ada program library manager will use DISK and not DBA3: when referencing file and directory specifications.

For example, a library index file will reference DISK: rather than DBA3: for entered units. Then if DBA3: is swapped with another device, reassigning the logical name DISK to the new device will make the entered references correct.

5.7.1.2 Using Rooted Directory Syntax

Rooted directory syntax allows programs and utilities to refer to a device and a directory tree as a logical device and a top-level directory. A rooted directory is a concealed-device logical name that defines both a hidden device name and a hidden root directory. Once a rooted directory has been defined, all subsequent directory references will refer to the root directory or any of the directories in the directory tree below the root directory.

To define a rooted directory, you must use the DCL DEFINE or ASSIGN commands with the /TRANSLATION_ATTRIBUTES=CONCEALED qualifier. In the following example, the rooted directory BASE is defined as the directory DBA3:[PROJ.HOTEL.]. Note the trailing period (.) in the directory specification.

```
$ DEFINE/TRANSLATION_ATTRIBUTES=CONCEALED BASE DBA3:[PROJ.HOTEL.]
```

You can then refer to subdirectory DBA3:[PROJ.HOTEL.ADALIB] using the rooted directory syntax BASE:[ADALIB]. The device (DBA3:) and the directory structure ([PROJ.HOTEL]) are hidden when you use that syntax. In other words, the root directory, BASE, behaves as a top-level directory. For example:

```
$ ACS SET LIBRARY BASE:[ADALIB]
```

5.7.2 Copying Program Libraries

NOTE

When copying program libraries, remember that other libraries may reference them for entered units. To reference the new locations of copied libraries, you need to use the ACS ENTER UNIT/REPLACE command, specifying the new library location. If you did not originally use rooted directories to refer to the entered units, the ACS REENTER command reenters the units from their original libraries.

The best method for copying a program library from one device or directory to another is to use the VMS Backup Utility; see Section 5.7.3.

Another method is to create the new directory and then use the DCL COPY command. However, note the following restrictions:

- You cannot use this method across DECnet; if you are copying libraries across DECnet, use the Backup Utility.
- The directory to which you are copying the library must be empty.
- When copying a tree of sublibraries, you can use the DCL COPY command only to copy the top program library. If you use the DCL COPY command to copy a sublibrary, the copied sublibrary points to its original parent library, unless you have used a rooted directory.
- You may run into problems with the file creation dates that the DCL COPY command assigns to the files it copies.

A third way to copy a program library is to create a new program library using the ACS CREATE LIBRARY or CREATE SUBLIBRARY command, and then to use the ACS COPY UNIT and ENTER UNIT commands to copy and enter units into the new program library. If units have been entered from several program libraries, this method requires more individual operations than the backup or DCL COPY command methods. For example:

```
ACS> CREATE LIBRARY USER:[JONES.NEW.ADALIB]
ACS> SET LIBRARY USER:[JONES.NEW.ADALIB]
ACS> COPY UNIT DISK:[SMITH.LISTS.ADALIB] unit-name[,...]
   .
   .
   .
ACS> ENTER UNIT program-library1 unit-name[,...]
ACS> ENTER UNIT program-library2 unit-name[,...]
   .
   .
   .
```

5.7.3 Backing Up and Restoring Program Libraries

To back up program libraries, use the VMS Backup Utility. For example, the following command copies a library tree from one set of directories to another set of directories on the same disk and node:

```
$ BACKUP USER:[JONES.HOTEL...] USER:[JONES.NEW_HOTEL...]
```

The following command backs up a library from a set of directories on the local node and transfers the save-set across DECnet to the node CENTRL:

```
$ BACKUP USER:[JONES.HOTEL.ADALIB] -
_$ CENTRL"PROJ PASSWORD"::DISK:[PROJ.JONES]HOTEL_ADALIB.BCK -
_$ /SAVE_SET
```

The following command restores the save-set on node CENTRL:

```
$ BACKUP [PROJ.JONES]HOTEL_ADALIB.BCK/SAVE_SET -
_$ [PROJ.JONES.ADALIB]
```

See the *Guide to Maintaining a VMS System* and the *VMS Backup Utility Manual* for information on using the Backup Utility.

You can make your backups of library trees easier if you use concealed-device logical names and rooted directory syntax to make unit and parent library references directory independent. See Sections 5.7.1.1 and 5.7.1.2 for more information on concealed-device logical names and rooted directory syntax.

For example, consider the following sublibrary tree:

- The logical name TOP is assigned to the directory DBA3:[HOTEL.]:

  ```
  $ DEFINE/TRANSLATION_ATTRIBUTES=CONCEALED TOP DBA3:[HOTEL.]
  ```

- The sublibrary [JONES.HOTEL.SUBLIB] is created as a sublibrary of TOP:[ADALIB]:

  ```
  $ ACS CREATE SUBLIBRARY/PARENT=TOP:[ADALIB] [JONES.HOTEL.ADALIB]
  ```

If TOP is backed up and restored to another device or directory, reassignment of the logical name TOP will make the sublibrary point to the correct location:

```
$ DEFINE/TRANSLATION_ATTRIBUTES=CONCEALED TOP new-dev-or-dir-spec
```

You can also use rooted directory syntax to obtain device or directory independence for entered units. Then, if the new device or directory has been reassigned properly, you do not have to enter or reenter the units after a backup or restore operation to a different device or directory.

5.7.4 Reorganizing Program Libraries

Each time you compile or recompile one or more units, your program library is updated. If your program library is updated frequently, ACS command performance may degrade. To improve and optimize ACS command performance, enter the ACS REORGANIZE command. For example:

```
$ ACS REORGANIZE
```

By default, as shown in this example, the ACS REORGANIZE command reorganizes your current program library.

In general, you should consider reorganizing each of your program libraries frequently, especially after doing many compilations or recompilations into the same library.

5.7.5 Verifying and Repairing Program Libraries

The ACS VERIFY command performs consistency checks on library files and requires only read access to a program library. The ACS VERIFY/REPAIR command corrects certain kinds of errors and requires exclusive read-write access to a program library. Both commands operate on the current program library by default, or on a specified program library.

When you execute the ACS CHECK, COMPILE, RECOMPILE, or LINK command, you may encounter the following errors:

- Missing units
- Obsolete units
- Obsolete references to entered units
- Missing copied source files (in the case of the COMPILE and RECOMPILE commands)

You may occasionally receive other, unexpected, diagnostics with a program library—for example, messages about missing or corrupted files associated with units in the library. The COMPILE, RECOMPILE, or LINK commands may detect these errors; the CHECK command may not. If you suspect that program library files have been corrupted or are missing, you should enter the VERIFY command.

The VERIFY command checks the following items:

- The format of the library index file.

- Whether all files cataloged in the library index file exist in the program library and are accessible. In the case of entered units, the VERIFY command checks whether the files exist in the library from which they were entered.
- Whether all files that exist in the program library directory are cataloged in the library index file and have the correct format.
- Whether the protection code of cataloged files is consistent with that of the library index file (see Section 5.6.1 for information on protection codes).

Under normal conditions, the VERIFY command issues a success message. For example:

```
$ ACS VERIFY
%I, USER:[JONES.HOTEL.ADALIB] verified
```

If you use the /LOG qualifier, the VERIFY command issues a separate message for each unit defined in the program library, as well as a final summary message.

If inconsistencies exist, the VERIFY command issues error messages indicating the units or files that are inconsistent. The following example shows the kinds of conditions that the VERIFY command can detect (typically, these conditions should rarely occur):

```
$ ACS VERIFY [PROJ.ADALIB]
%E, Inconsistent file protection USER:[PROJ.ADALIB]SQR.OBJ;3
%E, error opening USER:[PROJ.ADALIB]TEST_STACKS.OBJ;21 as
%E, input file not found
%E, USER:[PROJ.ADALIB]ODD.COM;12 is not cataloged
     in library USER:[PROJ.ADALIB]
%E, USER:[PROJ.ADALIB]SCREEN_IO.ACU;7 is not cataloged
     in library USER:[PROJ.ADALIB]
%I, Units with inaccessible files are obsolete. If repair
    (VERIFY/REPAIR) is not possible, then recompilation of
    these units is necessary; after entering a VERIFY/REPAIR
    command, the CHECK command will show any obsolete units
%E, USER:[PROJ.ADALIB] has uncorrected errors
```

The messages in this example have the following meaning:

- The protection code of file SQR.OBJ;3 is inconsistent with that of the library index file.
- File TEST_STACKS.OBJ;21 is cataloged in the library index file but is not in the program library (the file is inaccessible).

- Files ODD.COM;12 and SCREEN_IO.ACU;7 are not cataloged in the library index file (these files do not belong in the program library directory).

These kinds of errors are not detected by the CHECK command, which you use to determine whether any units in a closure are missing or obsolete.

You can use the ACS VERIFY/REPAIR command to correct some of the errors reported by the VERIFY command. The VERIFY/REPAIR command performs the same checks as the VERIFY command, and takes corrective action on the specified program library, as follows:

- Identifies any files in the program library directory that are not cataloged in the library index file. Deletes any uncataloged files with a file type of .OBJ, .ACU, or .ADC. Deletes any uncataloged files with other file types only if you have also specified the /CONFIRM qualifier and given an affirmative response.
- As necessary, changes the file protection on .OBJ, .ACU, and .ADC files to be consistent with the protection code for the library index file.
- Marks as obsolete any unit whose .OBJ or .ACU file is inaccessible. A later VERIFY/REPAIR command will reset any such marks if the associated files are again available.
- Removes references to inaccessible copied source files (.ADC) from the library index file.
- Deletes any index entry with an illegal format from the library index file.

The VERIFY/REPAIR command does not take corrective action for entered units.

The VERIFY/REPAIR command requires exclusive read-write access to the program library to be verified—that is, you must first execute the SET LIBRARY/EXCLUSIVE command interactively (see Chapter 2) and then enter the VERIFY/REPAIR command (also interactively).

The following example shows the use of the VERIFY/REPAIR command with the error conditions reported by the VERIFY command in the preceding example. The /LOG qualifier lists the action taken for each unit or file being repaired (only units and files that had inconsistencies are shown in the example):

```
$ ACS
ACS> SET LIBRARY/EXCLUSIVE [PROJ.ADALIB]
ACS> VERIFY/REPAIR/LOG
    .
    .
    .
%E, Inconsistent file protection USER:[PROJ.ADALIB]SQR.OBJ;3
%W, SQR verified and repaired
    .
    .
    .
%E, error opening USER:[PROJ.ADALIB]TEST_STACKS.OBJ;21 as
-E, input file not found
%W, TEST_STACKS verified and repaired
    .
    .
    .
%E, USER:[PROJ.ADALIB]ODD.COM;12 is not cataloged
     in library USER:[PROJ.ADALIB]
%E, USER:[PROJ.ADALIB]SCREEN_IO.ACU;7 is not cataloged
     in library USER:[PROJ.ADALIB]
%I, Units with inaccessible files are obsolete. If repair
    (VERIFY/REPAIR) is not possible, then recompilation of
    these units is necessary; after entering a VERIFY/REPAIR
    command, the CHECK command will show any obsolete units
%E, USER:[PROJ.ADALIB] has uncorrected errors
```

In this example, the VERIFY/REPAIR command has taken the following actions:

- Changed the protection of file SQR.OBJ;3 to be consistent with the protection of the library index file
- Marked the unit TEST_STACKS as obsolete, because its .OBJ file (TEST_STACKS.OBJ;21) is inaccessible
- Kept the uncataloged file ODD.COM;12, because its file type is not .OBJ, .ACU, or .ADC, and because the /CONFIRM qualifier was not used
- Deleted the uncataloged file SCREEN_IO.ACU;7, because its file type is .ACU

The following steps delete the uncataloged file ODD.COM;12:

```
ACS> VERIFY/REPAIR/CONFIRM
%E, error opening [PROJ.ADALIB]TEST_STACKS.OBJ as input
-E, file not found
%E, USER:[PROJ.ADALIB]ODD.COM;12 is not cataloged
     in library USER:[PROJ.ADALIB]
USER:[PROJ.ADALIB]ODD.COM;12, delete? [N]: y
%I, Units with inaccessible files are obsolete. If repair
    (VERIFY/REPAIR) is not possible, then recompilation of
    these units is necessary; after entering a VERIFY/REPAIR
    command, the CHECK command will show any obsolete units
%W, USER:[PROJ.ADALIB] verified and repaired
```

There are two ways to make the unit TEST_STACKS current:

- Because TEST_STACKS has been marked obsolete, you could use the RECOMPILE command. For example:

  ```
  ACS> RECOMPILE TEST_STACKS
  ```

- Alternatively, if a current copy of the missing file, TEST_STACKS.OBJ;21, is available in another program library, you could use the DCL COPY or BACKUP command to create a copy of the file in the program library [PROJ.ADALIB]. For example:

  ```
  $ COPY [backup-directory]TEST_STACKS.OBJ;21 [PROJ.ADALIB]
  ```

After the TEST_STACKS.OBJ file has been copied to [PROJ.ADALIB], the VERIFY/REPAIR command must be reentered so that TEST_STACKS can be marked as current.

5.7.6 Recompiling Units After a New Release or Update of VAX Ada

When an update or full release of VAX Ada is installed on your system, previously compiled units, as well as references to entered units, may be rendered obsolete. Your system manager should inform you of this condition, which will, in any case, become evident when you try to use the program library manager or the compiler with obsolete units.

To make the contents of your program libraries current, you need to perform the following steps for each of your program libraries. Note that a program library with entered units needs to be made current *after* the entered units are made current in their own libraries. The program library manager issues an error message if you try to recompile a unit that depends on an obsolete entered unit.

1. Use the ACS SET LIBRARY command to define the current program library:

   ```
   $ ACS SET LIBRARY [JONES.HOTEL.ADALIB]
   ```

2. Use the ACS REENTER command to reenter current references to the VAX Ada predefined units from the current program library:

```
$ ACS REENTER *
```

   Consult the cover letter and release notes supplied with the release of VAX Ada that you are using. If new units have been added to the VAX Ada predefined units and you want to enter them into the current program library, use the ACS ENTER UNIT command, specifying the appropriate unit names:

```
$ ACS ENTER UNIT ADA$PREDEFINED unit-name[,...]
```

3. Use the ACS RECOMPILE command to make current all units in the current program library:

```
$ ACS RECOMPILE *
```

5.8 Working with Multiple Targets

When working with multiple targets (for example VMS and VAXELN targets), you need to know which parts of your code are target-specific and which are target-dependent. You also need to know how big an effect a change in the target can have. The following sections discuss topics related to program portability and target dependence.

5.8.1 Determining VAX Ada Program Portability

To determine if your VAX Ada program uses certain potentially nonportable features, you can enter the ACS SHOW PROGRAM/PORTABILITY command or you can use the /SHOW=PORTABILITY qualifier with any of the VAX Ada compilation commands (DCL ADA or ACS COMPILE or RECOMPILE). The /SHOW=PORTABILITY qualifier (which is the default for all of the compilation commands) causes the compiler to include a portability summary in the compilation listing file (the /LIST qualifier must also be specified).

The following sections discuss the factors affecting the portability of a VAX Ada program, and identify those features that may appear in the portability summary.

5.8.1.1 Factors Affecting Portability

A program's portability depends on the set of available implementations that are appropriate for the program. For example, the Ada Standard does not specify the range of digits for floating-point types that must be supported by an implementation. Thus, the following type declaration may or may not be portable to all relevant implementations:

```
type REAL is digits 9;
```

If an implementation can support the requested accuracy and implied range, then the program should be portable with respect to that implementation. If the implementation cannot support the requested accuracy, then it will produce an error during compilation (rather than allowing the program to compile and then execute with unacceptable results). The use of an implicit underlying type—in this case, the VAX Ada predefined type LONG_FLOAT (and either a D_floating or G_floating representation)—is not relevant to whether or not the program is portable.

The explicit use of a predefined type, such as LONG_FLOAT, also may or may not be portable. For example:

```
type REAL is new LONG_FLOAT;
```

The fact that some other implementation may support a predefined type LONG_FLOAT (as described in the Ada Standard) does not ensure that your program is portable to that implementation. In particular, the accuracy provided by that implementation may be less than the accuracy provided by VAX Ada—which may or may not be significant to your program.

The VAX Ada portability summary does not list implicit uses of the type LONG_FLOAT (as in the first example declaration of the type REAL); it does list explicit uses of the type LONG_FLOAT (as in the second declaration).

The abstraction properties of the Ada language imply that even when a particular construct is defined by a nonportable construct, uses of that particular construct are not necessarily also nonportable. For example, an unchecked conversion from the type INTEGER to the type ADDRESS could be implemented across a large number of Ada implementations in various ways—but it is the conversion declaration, not the conversion call, that you should examine when porting a program that uses the conversion function.

Another example of this concept occurs in the VAX Ada implementation of the predefined package TEXT_IO. The private part of TEXT_IO's specification uses some implementation-defined pragmas, such as the pragma IMPORT_PROCEDURE. The package body uses even more nonportable

constructs, such as the type ADDRESS, the implementation-defined attribute TYPE_CLASS, and other VAX Ada-specific features. However, the portability of programs that use the package TEXT_IO is not compromised.

Another consideration is that pragmas (which, as required by the Ada Standard, cannot affect the legality of a program) may or may not be relevant to the correct operation and/or portability of a program. For example, a program may work correctly only if the pragma SHARED is supported by an implementation, or only if the pragma PRIORITY is supported with a certain range of priorities. For this reason, the portability summary shows the use of many of the language-defined pragmas as well as the use of all of the implementation-defined pragmas.

5.8.1.2 Features Listed in the Portability Summary

The portability summary lists one or more of the features or constructs shown in Table 5–3. The summary briefly describes each feature or construct, and each description is followed by the line numbers where each use of the feature or construct occurs. Features or constructs that are implementation-specific are marked with an asterisk (*). For example:

```
 . . .

PORTABILITY SUMMARY

predefined SHORT_INTEGER or SHORT_SHORT_INTEGER
                                    3
predefined LONG_FLOAT or LONG_LONG_FLOAT
                                    4
with SYSTEM                         1
predefined ADDRESS                  5
predefined NON_ADA_ERROR*           9
attribute ADDRESS                   7

        where * indicates an implementation-defined feature
 . . .
```

Italicized text is used in Table 5–3 to explain some of the features or constructs; the text does not appear in the actual portability summary.

Whether or not you specify the /SHOW=PORTABILITY qualifier for a compilation, the use of any of these features is always recorded (without specific line numbers) in the current program library. You can obtain portability information at any time with the ACS SHOW PROGRAM/PORTABILITY command.

Table 5–3: Features or Constructs that May Appear in a Portability Summary

Implementation-Defined Types in the Package STANDARD
predefined SHORT_INTEGER or SHORT_SHORT_INTEGER
predefined LONG_FLOAT or LONG_LONG_FLOAT *(that is, explicit rather than implicit uses of these types, as discussed previously)*

Entities in the Predefined Package SYSTEM
with SYSTEM *(that is, use of predefined SYSTEM in a **with** clause)*
predefined NAME *(includes type NAME and any of its enumerals)*
predefined named number (such as MAX_INT)
predefined PRIORITY
predefined F_FLOAT, D_FLOAT, G_FLOAT or H_FLOAT*
predefined ADDRESS *(includes type ADDRESS and constant ADDRESS_ZERO)*
instantiation of FETCH_FROM_ADDRESS or ASSIGN_TO_ADDRESS*
predefined TYPE_CLASS* *(includes type TYPE_CLASS and any of its enumerals)*
predefined AST_HANDLER* *(includes type AST_HANDLER and constant NO_AST_HANDLER)*
predefined NON_ADA_ERROR*
predefined type, subtype, or special operator for VAX storage (such as UNSIGNED_LONGWORD)*
predefined conversion for VAX storage (such as TO_BIT_ARRAY_32)*
predefined read or write input-output register*
predefined read or write processor register*
predefined ALIGNED_WORD*
predefined add, set, or clear interlocked*
predefined INSQ_STATUS or REMQ_STATUS*
predefined insert or remove queue interlocked*

(continued on next page)

Table 5–3 (Cont.): Features or Constructs that May Appear in a Portability Summary

Predefined Procedure UNCHECKED_DEALLOCATION
with UNCHECKED_DEALLOCATION *(that is, use of predefined UNCHECKED_DEALLOCATION in a **with** clause)*
instantiation of UNCHECKED_DEALLOCATION
Predefined Function UNCHECKED_CONVERSION
with UNCHECKED_CONVERSION *(that is, use of predefined UNCHECKED_CONVERSION in a **with** clause)*
instantiation of UNCHECKED_CONVERSION
Representation Clauses
address representation clause
enumeration representation clause
length SIZE representation clause
length STORAGE_SIZE representation clause
length SMALL representation clause
record representation clause
Attributes
attribute ADDRESS
attribute AST_ENTRY*
attribute BIT*
attribute MACHINE_SIZE*
attribute NULL_PARAMETER*
attribute SIZE
attribute STORAGE_SIZE
attribute TYPE_CLASS*

(continued on next page)

Table 5–3 (Cont.): Features or Constructs that May Appear in a Portability Summary

Pragmas
unknown pragmas (*that is, any pragma not recognized by VAX Ada*)
unsupported pragmas (*that is, any pragma supported by another Ada implementation*)
pragma AST_ENTRY*
pragma EXPORT_EXCEPTION*
pragma EXPORT_FUNCTION*
pragma EXPORT_OBJECT*
pragma EXPORT_PROCEDURE*
pragma EXPORT_VALUED_PROCEDURE*
pragma IDENT*
pragma IMPORT_EXCEPTION*
pragma IMPORT_FUNCTION*
pragma IMPORT_OBJECT*
pragma IMPORT_PROCEDURE*
pragma IMPORT_VALUED_PROCEDURE*
pragma INTERFACE
pragma INLINE_GENERIC*
pragma LONG_FLOAT*
pragma MAIN_STORAGE*
pragma MEMORY_SIZE
pragma PACK
pragma PRIORITY
pragma PSECT_OBJECT
pragma SHARED
pragma SHARE_GENERIC
pragma STORAGE_UNIT
pragma SUPPRESS
pragma SUPPRESS_ALL*

(continued on next page)

Table 5–3 (Cont.): Features or Constructs that May Appear in a Portability Summary

Pragmas
pragma SYSTEM_NAME
pragma TASK_STORAGE*
pragma TIME_SLICE*
pragma TITLE*
pragma VOLATILE*

5.8.2 Setting the System Name

The VAX Ada program library manager, as the interface to the VAX Ada compiler and VMS Linker, is sensitive to differences in targets through the value of the predefined constant SYSTEM_NAME in the package SYSTEM. This constant can have a value of either VAXELN or VAX_VMS.

The value of SYSTEM.SYSTEM_NAME does not cause the compiled code to differ. It is used to determine target-related compilation unit dependences, which can occur in your Ada code in the following cases:

- Use of SYSTEM.SYSTEM_NAME causes either a VAX_VMS or a VAXELN dependence.
- Use of the pragma TIME_SLICE causes a VAX_VMS dependence.
- Use of the pragma AST_ENTRY or the AST_ENTRY attribute causes a VAX_VMS dependence.
- Use of any of the relative or indexed input-output packages causes a VAX_VMS dependence.
- Use of the package VAXELN_SERVICES causes a VAXELN dependence.

For example, if a compilation unit uses the pragma AST_ENTRY and the system name at compile time is VAXELN, you are warned that your unit depends on SYSTEM.SYSTEM_NAME and that the pragma AST_ENTRY is ignored for a VAXELN target. Similarly, if a unit uses the AST_ENTRY attribute and the system name at compile time is VAXELN, you are warned that your unit depends on SYSTEM.SYSTEM_NAME and that your use of the AST_ENTRY attribute is illegal.

When you create a program library or sublibrary, the default value of SYSTEM.SYSTEM_NAME is VAX_VMS. You can use the /SYSTEM_NAME qualifier on the ACS CREATE LIBRARY or CREATE SUBLIBRARY command to explicitly determine the value of SYSTEM.SYSTEM_NAME, or you can permanently set the system name to VAXELN (or set it back to VAX_VMS) by performing one of the following operations:

- Compiling the predefined Ada pragma SYSTEM_NAME
- Executing the ACS SET PRAGMA command (ACS SET PRAGMA/SYSTEM_NAME=VAX_VMS or ACS SET PRAGMA/SYSTEM_NAME=VAXELN)

To determine the current setting for your current program library, use the ACS SHOW LIBRARY/FULL command; to determine system-name dependences for individual program units, use the ACS SHOW PROGRAM command.

You can temporarily override the current setting when you link or export units by using the /SYSTEM_NAME qualifier on the ACS LINK and EXPORT commands. For example, if you are working in a VMS environment (SYSTEM.SYSTEM_NAME=VAX_VMS), and the units you have compiled do not contain any of the VMS-specific features, you can link them for a VAXELN target with the ACS LINK/SYSTEM_NAME=VAXELN command. However, a link-time error occurs if a unit depends on the value of SYSTEM.SYSTEM_NAME and a /SYSTEM_NAME qualifier specifies a different value. See Chapter 4 for more information on the ACS LINK and EXPORT commands.

When you use the pragma SYSTEM_NAME or the ACS SET PRAGMA command to change the system name (either with an argument of VAX_VMS or VAXELN), an implicit recompilation of the package SYSTEM occurs. Those units that depend on the value of SYSTEM.SYSTEM_NAME are then made obsolete, and must be recompiled in the context of the new system name. For example, consider the following program (dashed lines separate the individual compilation units):

```
procedure TASK_WORK is          -- VMS-dependent procedure.
   pragma TIME_SLICE(0.4);
   task type T;
   type TASK_FORCE_TYPE is
      array (INTEGER range 1..5) of T;
   TASK_FORCE: TASK_FORCE_TYPE;
```

```
   task body T is separate;    -- Task body is a subunit.
begin
    . . .
end TASK_WORK;
-------------------------------------------------------------------
with TASK_WORK;
procedure ALL_WORK is          -- Main program, depends on
                               -- target-dependent TASK_WORK.
begin
    . . .
   TASK_WORK;
    . . .
end ALL_WORK;
-------------------------------------------------------------------
with TEXT_IO; use TEXT_IO;
separate (TASK_WORK)
task body T is                 -- Target-independent subunit depends
                               -- on target-independent package
                               -- TEXT_IO and target-dependent
                               -- ancestor, TASK_WORK.
begin
   PUT_LINE ("My work's just starting...");
    . . .
   delay 3.0;
    . . .
   PUT_LINE ("My work's all done!");
end T;
```

If you compile these units into a program library for which SYSTEM.SYSTEM_NAME equals VAX_VMS, and subsequently use the ACS SET PRAGMA command to set SYSTEM_NAME to VAXELN, then the following effects will occur:

- Procedure TASK_WORK becomes obsolete because it depends on SYSTEM.SYSTEM_NAME=VAX_VMS.
- The main program, ALL_WORK, becomes obsolete because it depends on procedure TASK_WORK.
- The subunit TASK_WORK.T becomes obsolete because it depends on its ancestor, TASK_WORK.

All three units would have to be recompiled before they could be linked, and recompilation would result in a warning because the pragma TIME_SLICE is ignored for VAXELN targets. Chapter 1 discusses unit dependences in more detail.

Chapter 6

Debugging VAX Ada Programs

A debugger is a tool that helps you locate run-time errors. You use it with a program that has been compiled and linked successfully, but does not run correctly. For example, the output may be wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively so that you can locate the point at which the program stopped working correctly, and then you can examine the state of the program at that point.

This chapter is an introduction to using the VMS Version 5.0 Debugger with VAX Ada programs. This chapter provides you with the following information:

- An overview of the debugger
- Information to get you started using the debugger with Ada programs
- Ada-specific debugger information
- A sample terminal session that demonstrates using the debugger

You can obtain additional debugger information from the following sources:

- Online HELP is available during debugging sessions (type HELP at the debugger prompt).
- Chapter 7 describes the techniques available for debugging multitasking VAX Ada programs.
- Appendix B lists debugger commands by function.
- The *VMS Debugger Manual* provides complete reference information on the VMS Debugger and its commands.

6.1 VMS Debugger Overview

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

The debugger recognizes the syntax, expressions, data typing, and other constructs of VAX Ada. The debugger also supports the following VAX languages:

BASIC	DIBOL	PL/I
BLISS	FORTRAN	RPG II
C	MACRO-32	SCAN
COBOL	Pascal	

If your program is written in more than one VAX language, you can change from one language to another during a debugging session. The current source language determines the format used for entering and displaying data, as well as other features that have language-specific settings (for example, operators and operator precedence, and case sensitivity or insensitivity).

By entering debugger commands at your terminal, you can perform the following operations:

- Start, stop, and resume the program's execution.
- Trace the execution path of the program.
- Monitor s lected locations, variables, or events, including Ada exceptions and tasks.
- Examine and modify the contents of variables, or force events to occur.

After you find the error in your program, you can edit the source code and compile, link, and run the corrected version.

The debugger also provides the following features to help you debug your programs:

- **Online HELP**—Online HELP is always available during a debugging session and contains information on all the debugger commands and also information on selected topics.
- **Source Code Display**—You can display lines of source code at any time during a debugging session, without affecting program execution.

- **Screen Mode**—You can display various kinds of information in scrollable windows, which can be moved around the screen and resized. Automatically updated source, instruction, and register displays are available. You can selectively direct debugger input, output, and diagnostic messages to displays.
- **Keypad Mode**—When you invoke the debugger, several commonly used debugger command sequences are assigned by default to the keys of the numeric keypad (if you have a VT100, VT52, or LK201 keypad).
- **Source Editing**—As you find errors during a debugging session, you can use the debugger EDIT command to invoke any editor available on your system. (You can first specify the editor you want with the debugger SET EDITOR command.)
- **Command Procedures**—The debugger allows you to execute a command procedure to re-create a debugging session, to continue a previous session, or to avoid typing the same debugger commands many times during a debugging session.
- **Symbol Definitions**—You can define your own symbols to represent lengthy commands, address expressions, or values.
- **Initialization Files**—You can create an initialization file containing commands to set your default debugging modes, screen display definitions, keypad key definitions, symbol definitions, and so on. In addition, you may want to have special initialization files for debugging specific programs.
- **Log Files**—You can record the commands you enter during a debugging session and the debugger's responses to those commands in a log file. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

6.2 Getting Started with the Debugger

The following sections explain how to prepare your Ada program for debugging, and then focus on the tasks required to run an Ada debugging session. For more detailed information that is not Ada-specific, see the *VMS Debugger Manual*.

6.2.1 Compiling and Linking a Program to Prepare for Debugging

Before you can use the debugger, you must compile and link your program as explained in this section. The following example shows how to compile and link a VAX Ada program before using the debugger. The program consists of a single compilation unit named ADD_INTEGERS:

```
$ ADA/DEBUG/NOOPTIMIZE ADD_INTEGERS
$ ACS LINK/DEBUG ADD_INTEGERS
```

The /DEBUG qualifier on the DCL ADA command (also the default) causes the compiler to write the debugger symbol records associated with the compilation unit ADD_INTEGERS into the object module in the current Ada program library. These records allow you to use the names of variables and other symbols declared in debugger commands. They also allow the debugger to display source code for the program.

The /NOOPTIMIZE (or /OPTIMIZE=NONE) qualifier prevents program optimizations, which may make program symbols inaccessible to the debugger, or which may make the generated code difficult to understand. You should use the /NOOPTIMIZE qualifier when you compile a program to prepare it for debugging; the default is /OPTIMIZE. After you debug your program, you can compile it again with the /OPTIMIZE qualifier. The /OPTIMIZE=DEVELOPMENT option is another alternative that provides some optimization, but also prepares the code for debugging.

See Chapter 3 and Appendix A for more information on the /[NO]OPTIMIZE qualifier; see Chapter 3 for more information on recompiling your program with different qualifiers.

If your program has several compilation units, you must compile all of the units that you want to debug with the appropriate qualifiers. For example:

```
$ ADA/DEBUG/NOOPTIMIZE SUM, MAIN
$ ACS LINK/DEBUG MAIN
```

Note the following points when compiling Ada programs or units for debugging:

- The DCL ADA and ACS COMPILE and RECOMPILE commands assume the /DEBUG qualifier by default. Unless you use the /NODEBUG qualifier with these commands, they will automatically compile units for debugging.
- The VAX Ada predefined units in the ADA$PREDEFINED program library on your system have been compiled with the /NODEBUG qualifier. Before using the debugger to refer to names declared in the predefined units, you must first copy the predefined unit source files

using the ACS EXTRACT SOURCE command. Then, you must compile the copies into the appropriate library with the /DEBUG qualifier, and relink the program with the /DEBUG qualifier.

- If you use the /NODEBUG qualifier with one of the Ada compilation commands, only global symbol records are included in the modules for debugging. Global symbols in this case are names that the program exports to modules in other languages by means of the VAX Ada export pragmas: EXPORT_PROCEDURE, EXPORT_VALUED_PROCEDURE, EXPORT_FUNCTION, EXPORT_OBJECT, EXPORT_EXCEPTION, and PSECT_OBJECT.

The /DEBUG qualifier on the ACS LINK command causes the linker to include all debugging information in the closure of the specified unit in the executable image. This qualifier also causes the VMS image activator to start the debugger at run time. See Chapter 4 and Appendix A for more information on the ACS LINK command.

With the default /TRACEBACK qualifier and without the /DEBUG qualifier (or with the default /NODEBUG qualifier), the ACS LINK command includes only traceback information in the image—that is, it includes the names of compilation units, subprograms, and compiler-generated line numbers. Traceback information is used by the VMS traceback utility to identify the active calls and PC (program counter) location when an error occurs.

6.2.2 Starting and Ending a Debugging Session

To invoke the debugger, enter the DCL RUN command. The following messages will appear on your terminal:

```
$ RUN ADD_INTEGERS

         VAX DEBUG Version V5.0-00

%DEBUG-I-INITIAL, language is ADA, module set to ADD_INTEGERS
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
DBG>
```

The INITIAL message indicates that the debugging session is initialized for a VAX Ada program and that the name of the main program unit is ADD_INTEGERS. The initialization sets up any language-dependent debugger parameters.

The NOTATMAIN message indicates that execution is suspended before the elaboration of library units and before the start of the main program, so that you can execute this elaboration code under debugger control. This message

is issued both for Ada main programs and for non-Ada main programs that call Ada exported units.

Typing the GO command places you at the start of the main program. At that point, type the GO command again to start program execution. Execution continues until it is forced to pause or stop (for example, if the program prompts you for input, a breakpoint is triggered, or an error occurs).

The following message indicates that your program has completed successfully:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
    completion'
DBG>
```

If you want to continue debugging at this point, type EXIT and then enter the DCL RUN command to start a new debugging session.

To interrupt a debugging session and return to DCL level, press CTRL/Y. For example, pressing CTRL/Y is useful if your program loops or you want to interrupt a debugger command that is still in progress. See Chapter 7 for information on using CTRL/Y if your program has tasks.

To resume the debugging session after a CTRL/Y interruption, enter either the CONTINUE or DEBUG command at DCL level. Use the CONTINUE command to resume execution at the point which you interrupted the debugging session. If you interrupted the session because of an infinite loop, use the DEBUG command instead. The DEBUG command returns you to the debugger prompt so that you can enter another command. For example:

```
DBG> GO
   .
   .
   .
(infinite loop)
 CTRL/Y
Interrupt

$ DEBUG
DBG>
```

If you do not want to continue the debugging session, you can enter the DCL EXIT or STOP commands, or any DCL command that executes an image, such as COPY, MAIL, or RUN. The EXIT command is preferable to the STOP command because it invokes exit handlers in your program and the debugger, and thus causes orderly termination of the debugging session. The STOP command prevents exit handlers in your program or the debugger from executing, and thus is not recommended. Running another image is equivalent to entering the EXIT command and then running the image.

To suspend a debugging session to execute other images (for example, read MAIL or submit a batch job), enter the debugger SPAWN or ATTACH command. These debugger commands behave just like their DCL counterparts (see the *VMS DCL Dictionary*).

To end a debugging session, enter the EXIT command or press CTRL/Z:

```
DBG> EXIT
$
```

If you have linked an image with the /DEBUG qualifier, and you want to execute that image without the debugger and without relinking, you can use the DCL RUN command with the /NODEBUG qualifier. For example:

```
$ RUN/NODEBUG ADD_INTEGERS
```

The RUN/NODEBUG command is convenient for checking a program once you think it is error free.

6.2.3 Entering Debugger Commands

The debugger provides a comprehensive set of commands to help you debug your program. A summary by function appears in Appendix B; see the online HELP file and the *VMS Debugger Manual* for more information on individual commands.

You can enter debugger commands any time you see the debugger prompt (DBG>). Type the command on the keyboard and press the RETURN key. Note that if a command results in an informational-level error, the debugger executes the command, even though it issues a diagnostic message.

You can enter several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the previous line with a hyphen (-).

```
DBG> EXAMINE LONG_VARIABLE_NAME1, -
_DBG> LONG_VARIABLE_NAME2
```

If you have a Digital keyboard, you can use the numeric keypad to enter certain commands. Figure 6–1 identifies the predefined key functions. You can also redefine key functions with the DEFINE/KEY command.

Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE. To obtain a key's DEFAULT function, press the key. To obtain its GOLD function, first press the PF1 (GOLD) key, and then the key. To obtain its BLUE function, first press the PF4 (BLUE) key, and then the key.

In Figure 6–1, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline, from top to bottom, respectively. For example, pressing keypad key 0 enters the STEP command; pressing key PF1 and then key 0 enters the STEP/INTO command; pressing key PF4 and then key 0 enters the STEP/OVER command.

Enter the command HELP KEYPAD to get help on the keypad key definitions.

6.2.4 Viewing Your Source Code

The debugger provides two modes for displaying information: noscreen mode and screen mode. By default, when you invoke the debugger, you are in noscreen mode. But you may find that it is easier to view your source code in screen mode. Both modes—and some general considerations about displaying source code—are briefly described in the following sections.

Note that to view your source code, you must have compiled and linked the code with the /DEBUG qualifier in effect (see Section 6.2.1).

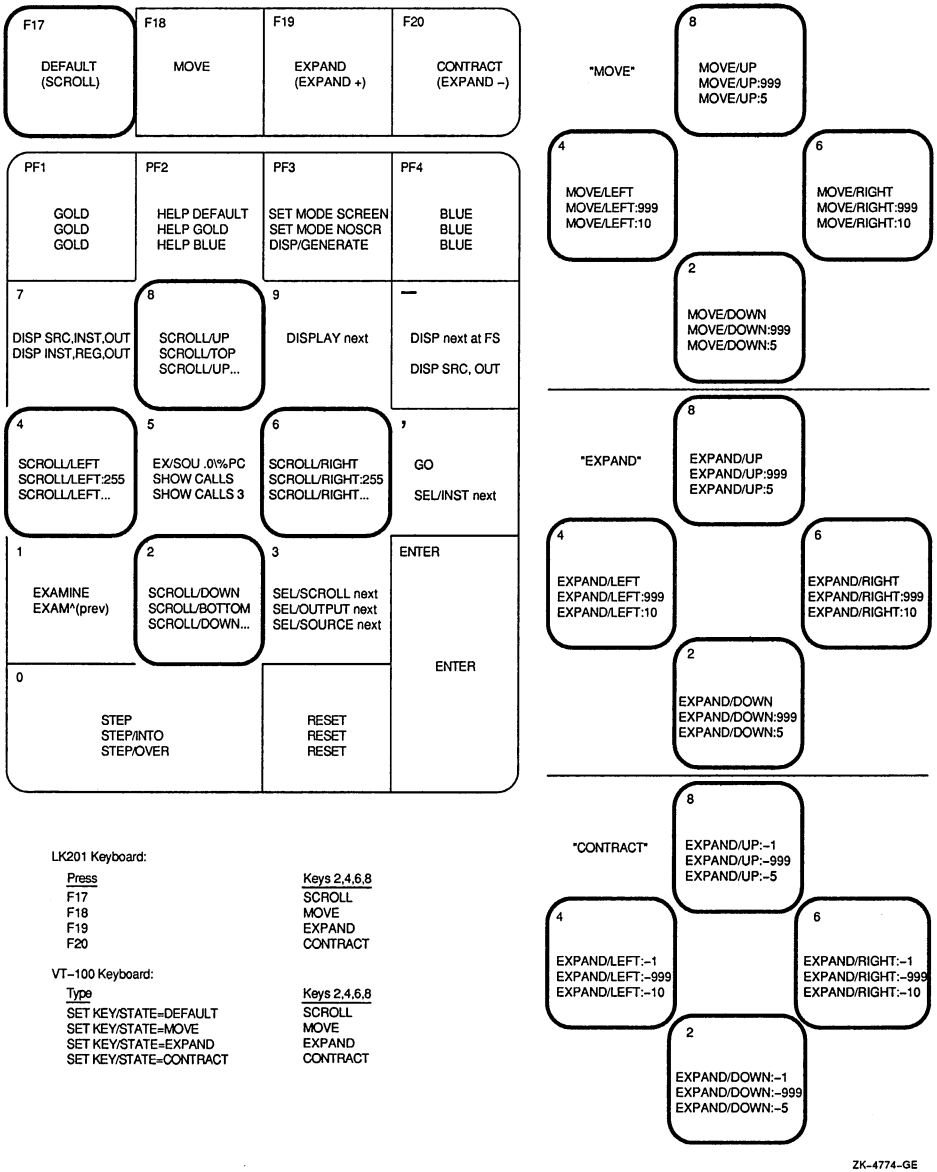
6.2.4.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. To invoke noscreen mode from screen mode, press the keypad key sequence PF1(GOLD key)-PF3 or type SET MODE NOSCREEN. See the sample debugging session in Section 6.7 for a demonstration of noscreen mode.

To display one or more source lines in noscreen mode, use the debugger TYPE command or the debugger EXAMINE/SOURCE command. For example, the following command displays line 9 of the module whose code is executing:

```
DBG> TYPE 9
module MAIN
     9:     I := 7;
DBG>
```

Figure 6–1: Debugger Keypad Key Functions



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT–100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

Similarly, the following command displays the next line to be executed, which corresponds to the current value of the program counter (PC):

```
DBG> EXAMINE/SOURCE .PC
module MAIN
     2: procedure MAIN is
DBG>
```

The display of source lines using the TYPE and EXAMINE/SOURCE commands is independent of program execution. To display source code from a module other than the one whose code is currently executing, use the TYPE command with a path name to specify the module. For example, the following command displays lines 2 through 6 of module MAIN:

```
DBG> TYPE MAIN\2:6
module MAIN
     2: procedure MAIN is
     3:    I,J,K: INTEGER;
     4:    function AVERAGE (Y: INTEGER) return INTEGER is
     5:    begin
     6:       return SUM(Y)/Y;
```

Note that the debugger also automatically displays source lines when it suspends execution at a breakpoint or watchpoint or after a STEP command, or when a tracepoint is triggered (see Section 6.3).

If the debugger cannot locate source lines for display, it issues a diagnostic message. Source lines may not be available for a variety of reasons. See Section 6.2.4.3 for more information.

6.2.4.2 Screen Mode

To invoke screen mode, press keypad key PF3 or enter the debugger SET MODE SCREEN command. In screen mode, the debugger splits the screen into three displays named SRC, OUT, and PROMPT, by default.
For example:

```
- SRC: module MAIN -scroll-source-----------------------
     1: with SUM;
     2: procedure MAIN is
     3:    I,J,K: INTEGER;
     4:    function AVERAGE (Y: INTEGER) return INTEGER is
     5:    begin
     6:       return SUM(Y)/Y;
     7:    end AVERAGE;
     8: begin
     9:   I := 7;
->  10:   J := SUM(I);
    11:   K := AVERAGE(J);
- OUT -output-------------------------------------------
stepped to MAIN.%LINE 10
MAIN.I: 7
MAIN.J: 0
MAIN.K: 214658404

- PROMPT -error-program-prompt----------------------------
DBG> STEP 2
DBG> EXAMINE I,J,K
DBG>
```

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

The OUT display, in the middle of the screen, shows the debugger's output in response to the commands that you enter.

The PROMPT display, at the bottom of the screen, shows the debugger prompt (DBG>), your input, debugger diagnostic messages, and program output. In the example, the two debugger commands that were entered (STEP 2 and EXAMINE I,J,K) are displayed. (The unpredictable values reported by the debugger for J and K indicate that lines 10 and 11 have not been executed yet; lines 10 and 11 will subsequently assign the values 4 to J and 14 to K.)

You can scroll the SRC and OUT displays to display information beyond the window's edge. Press keypad key 8 to scroll up and keypad key 2 to scroll down; press keypad key 6 to scroll right and keypad key 4 to scroll left. Use keypad key 3 to change the display to be scrolled (by default, the SRC display is scrolled). You can move and resize the default displays with the debugger MOVE, DISPLAY, EXPAND, and CONTRACT commands. You can also use the debugger SET DISPLAY command to create displays that show specific information. Scrolling a display does not affect program execution.

If the debugger cannot locate source lines for display, it issues a diagnostic message. Source lines may not be available for a variety of reasons. See Section 6.2.4.3 for more information.

6.2.4.3 Source Code Display Considerations

If the /COPY_SOURCE qualifier (the default) was in effect when you compiled your program, the debugger obtains the displayed Ada source code from the copied source files located in the program library where the program was originally compiled. If you compiled your program with the /NOCOPY_SOURCE qualifier, the debugger obtains the displayed Ada source code from the external source files associated with your program's compilation units. See Chapter 3 or the description of the DCL ADA and ACS COMPILE commands in Appendix A for more information on copied source files and the /[NO]COPY_SOURCE qualifier.

The file specifications of the copied or external source files are embedded in the associated object files. For example, if you have used the ACS COPY UNIT command to copy units, or the DCL COPY or BACKUP command to copy an entire library, the debugger still searches the original program library for copied source files. If, after copying, the original units have been modified or the original library has been deleted, the debugger may not find the original copied source files. Similarly, if you have moved the external source files to another disk or directory, the debugger may not find them.

In such cases, use the debugger SET SOURCE command to locate the correct files for screen-mode display. You can specify a search list of one or more program library or source code directories where the debugger should look for the files.

For example, the following command line instructs the debugger to look for copied source files first in the current program library, and then in the program library DISK:[SMITH.SHARE.ADALIB] (ADA$LIB is the logical name that the program library manager equates to the current program library):

```
DBG> SET SOURCE ADA$LIB,DISK:[SMITH.SHARE.ADALIB]
```

The debugger SET SOURCE command does not affect the search list for the source files (.ADA) that the debugger fetches when you use the debugger EDIT command. To tell the debugger EDIT command where to look for your source files, use the debugger SET SOURCE/EDIT command (see Section 6.6.2).

The debugger SHOW SOURCE command displays the current search list for copied or external source files. The debugger CANCEL SOURCE command cancels the current search list.

If the debugger cannot locate source lines for the routine (subprogram) that is currently executing, it tries to display source lines in the next routine down on the call stack for which source lines are available. It then issues a message like one of the following:

```
%DEBUG-W-UNAOPNSRC, unable to open source file USER:[PROJ.ADALIB]
    RESERVATIONS.ADC;1
-RMS-E-FNF, file not found

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

%DEBUG-W-NOSRCLIN, no source line for address 000006C6
```

Source lines may not be available for the following reasons:

- The PC value is within a system routine or a shareable image routine for which no source code is available.
- The PC value is within Ada initialization or elaboration code, for which no source code is available.
- The PC value is within a routine that was compiled without the /DEBUG compiler command qualifier (or was compiled with the /NODEBUG qualifier).
- The PC value is within a routine whose module is not set (see Section 6.5.2 for an explanation of module setting).
- The copied source file is not in the program library where the unit was originally compiled (see the preceding discussion).
- The external source file is not where it was when the unit was originally compiled (see the preceding discussion).
- The source file has been modified since the executable image was generated, and the original copied source file or external source file no longer exists (see the preceding discussion).

6.3 Controlling and Monitoring Program Execution

The following sections discuss a number of topics related to continuing and monitoring program execution:

- Starting and resuming program execution with the GO command
- Stepping through the program's code with the STEP command
- Determining the current value of the program counter (PC) with the SHOW CALLS command
- Suspending program execution with breakpoints

- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

NOTE

The debugger refers to Ada subprograms as *routines*.

The debugger refers to Ada compilation units as *modules*. In some cases, to have access to the symbol records of an arbitrary compilation unit, you may need to use the SET MODULE command. To specify names and other symbols that are multiply defined, you may need to use *path names* or the SET SCOPE command. Path names and the SET MODULE and SET SCOPE commands are explained in Section 6.5.

6.3.1 Starting and Resuming Program Execution

Two basic debugger commands start or resume program execution: GO and STEP. The GO command starts execution. The STEP command executes one or more source lines or instructions.

6.3.1.1 The GO Command

The debugger GO command starts program execution, which continues until forced to stop. The GO command is used most often in conjunction with breakpoints, tracepoints, and watchpoints (described in Sections 6.3.3 through 6.3.5). If you set a breakpoint in the path of execution and then enter the GO command, execution is suspended at that breakpoint. If you set a tracepoint, the path of execution through that tracepoint is monitored. If you set a watchpoint, execution is suspended when the value of the watched variable changes.

When debugging an Ada program that involves library packages, you can use the GO command in combination with breakpoints to stop at individual package specifications and bodies and then step through their elaboration code. (See Section 6.3.6 for detailed information on debugging library packages.)

You can also use the GO command to test for an exception condition. If an exception condition occurs and is not handled by your program, the debugger takes control and displays the DBG> prompt so that you can enter commands. If you are using screen mode, the pointer in the source display indicates where execution stopped. You can use the debugger SHOW CALLS command (explained in Section 6.3.2) to identify the currently active routine calls (the call stack).

You can interrupt a GO command by first pressing CTRL/Y and then entering the DCL DEBUG command. You can then enter debugger commands, look at the source display, enter a SHOW CALLS command, and so on. For example, this action is useful if your program has an infinite loop.

6.3.1.2 The STEP Command

The debugger STEP command allows you to execute one or more source lines or instructions, or to execute the program to the next instruction of a particular kind, for example, to the next debugger CALL instruction. By default, the STEP command executes a single source line at a time. In the following example, the STEP command executes one line, reports the action ("stepped to . . . "), and displays the line number (9) and source code of the next line to be executed:

```
DBG> STEP
stepped to MAIN.%LINE 9
     9:    I := 7;
DBG>
```

Execution is now suspended at the first machine code instruction for line 9 of the unit MAIN. Note the *path name* MAIN.%LINE 9 in this example. The debugger uses path names to refer to symbols. (See Section 6.5.3.2 for more information on path names.)

The STEP command can execute a number of lines at a time. In the following example, the STEP command executes two lines:

```
DBG> STEP 2
stepped to MAIN.%LINE 11
    11:    K := AVERAGE(J);
DBG>
```

Only those source lines for which machine code instructions were generated by the compiler are regarded as executable lines by the debugger. The debugger skips over any other lines, for example, comment lines or a line that contains only the keyword **begin** or an uninitialized object declaration. In the case of a source line that has more than one Ada statement, the line number corresponds to the first statement on that line. Also, if a line contains more than one statement, the debugger executes all the statements on that line as part of the single step.

NOTE

If you use the STEP command at the start of an Ada debugging session, before going to the beginning of the program using the preset breakpoint for that purpose, you will step through compiler-generated initialization code. You should always use the GO

command when you start a debugging session. You should also use the GO command and breakpoints to get to the beginning of any library packages. (See Section 6.3.6 for more information on debugging library packages.)

You can specify different stepping modes, such as stepping by machine instruction rather than by line (SET STEP INSTRUCTION or STEP/INSTRUCTION). Also, by default, the debugger steps over called routines; each routine is executed, but execution is not suspended within it. Entering the SET STEP INTO or STEP/INTO command causes the debugger to suspend execution within called routines, as well as within the routine that is currently executing.

NOTE

Ada task entry calls are not the same as subprogram calls because task entry calls are queued and may not execute right away. If you try to step into a task entry call, the results may not be what you expect. See Chapter 7 for more information on debugging tasks.

You can determine the current STEP mode with the SHOW STEP command. For example:

```
DBG> SHOW STEP
step type: source, nosilent, by line,
           into routine calls
DBG>
```

6.3.2 Determining Where Execution is Suspended

The debuggger SHOW CALLS command is useful when you are unsure where execution is suspended during a debugging session (for example, after a CTRL/Y interruption).

The SHOW CALLS command displays a traceback that lists the sequence of calls leading to the routine in which execution is currently suspended. For each routine (beginning with the routine in which execution is suspended), the debugger displays the following information:

- The name of the module (compilation unit) that contains the routine
- The name of the routine
- The line number at which the call was made (or at which execution is suspended, in the case of the current routine)

- The corresponding PC addresses (the relative PC address from the start of the routine, and the absolute PC address of the program)

Some of the information in the display (such as the routine name and the line number) depends on whether or not the module for a routine has been set. An asterisk next to a module name indicates that the module has been set. See Section 6.5.2 for more information on module setting.

For example:

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

*SUM              SUM                    1    00000002  00000622
*MAIN             AVERAGE                6    0000001D  0000067D
*MAIN             MAIN                  11    00000027  000006B2
*ADA$ELAB_MAIN    ADA$ELAB_MAIN               00000009  00000609
                  LIB$INITIALIZE              00000054  0000070C
 SHARE$ADARTL                                 00000000  00035DB2
*ADA$ELAB_MAIN    ADA$ELAB_MAIN               0000001B  0000061B
                  LIB$INITIALIZE              0000002F  000006E7
DBG>
```

This example indicates the following sequence of calls:

- Execution is currently at line 1 of subprogram SUM (in unit SUM).
- Subprogram SUM was called from line 6 of subprogram AVERAGE (in unit MAIN).
- Subprogram AVERAGE was called from line 11 of subprogram MAIN (in unit MAIN).

The modules ADA$ELAB_MAIN and SHARE$ADARTL execute the elaboration and initialization code for the procedure MAIN.

6.3.3 Suspending Program Execution

The debugger SET BREAK command allows you to select *breakpoints*, which are locations at which program execution is suspended. When you reach a breakpoint, you can enter commands to examine the current values of variables, check the call stack, and so on.

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

NOTE

If the symbol on which you want to set a breakpoint is in a scope that is not visible to the debugger, you may have to set its module. See Section 6.5.2.4 for more information on the debugger SET MODULE command.

If you set a breakpoint at a location currently used as a tracepoint, the tracepoint is canceled in favor of the breakpoint. See Section 6.3.4 for more information on tracepoints.

In the following example, the SET BREAK command sets a breakpoint on the procedure SUM. The GO command then starts execution. When the procedure SUM is encountered, execution is suspended. The debugger reports that the breakpoint at SUM has been reached ("break at . . . "), displays the source line (1) where execution is suspended, and prompts you for another command. At this breakpoint, you could enter the EXAMINE command (discussed in Section 6.4.1) to check on the current value of X, step through the procedure SUM using the STEP command, and so on.

```
DBG> SET BREAK SUM
DBG> GO
    .
    .
    .
break at routine MAIN.SUM
     1: function SUM (X: INTEGER) return INTEGER is
DBG>
```

When using the SET BREAK command, you can specify program locations using various kinds of *address expressions* (for example, line numbers, subprogram names, instructions, virtual memory addresses, or byte offsets). With high-level languages like Ada, you typically use subprogram names, library package names, labels, or line numbers, possibly with path names to ensure uniqueness. (See Section 6.3.6 for information on naming library packages with the SET BREAK command; see Section 6.5.3.2 for more information on path names.)

You should specify subprogram names and labels as they appear in the source code. You can determine line numbers from either a source code display in the debugger or a listing file. When specifying a line number, use the prefix %LINE; otherwise, the debugger interprets the line number as a memory location. For example, the following command sets a breakpoint at line 11 of the module (compilation unit) in which execution is suspended; the debugger suspends execution when the PC value is at the start of line 11:

```
DBG> SET BREAK %LINE 11
DBG>
```

Note that you can set breakpoints only on lines that result in machine code instructions. The debugger warns you if you try to do otherwise (for example, if you try to set a breakpoint on a comment line). To set a breakpoint on a line number in a module other than the one whose code is currently executing, specify the module's name in a path name. For example:

```
DBG> SET BREAK SUM\%LINE 6
DBG>
```

You do not always need to specify a particular program location, such as line 6 or SUM, to set a breakpoint. You can set breakpoints on events, such as exceptions (see Section 6.3.7). You can also use the SET BREAK command with the /LINE qualifier (but no parameter) to break on every line, or with the /CALL qualifier to break on every CALL instruction, and so on. For example:

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

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

```
DBG> SET BREAK SUM_LOOP DO (EXAMINE TOTAL)
DBG> GO
    .
    .
    .
break at SUM.SUM_LOOP
     5:    for I in 1..X loop
SUM.TOTAL:       0
DBG>
```

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

```
DBG> SHOW BREAK
    .
    .
    .
breakpoint at SUM.LOOP$5.%LINE 6 MAIN.SUM
breakpoint at SUM.SUM_LOOP
   do (EXAMINE TOTAL)
DBG>
```

If any portion of your program was written in Ada, two breakpoints that are associated with Ada tasking-exception events are automatically established when you invoke the debugger. When you enter a SHOW BREAK command under these conditions, the following breakpoints are displayed:

```
DBG> SHOW BREAK
breakpoint on ADA Event "DEPENDENTS_EXCEPTION" for any value
breakpoint on ADA Event "EXCEPTION_TERMINATED" for any value
```

These breakpoints are equivalent to entering the following commands:

```
DBG> SET BREAK/EVENT=DEPENDENTS_EXCEPTION
DBG> SET BREAK/EVENT=EXCEPTION_TERMINATED
```

See Section 6.3.7 for more information on debugging Ada exceptions; see Chapter 7 for more information on debugging Ada tasking programs that involve exceptions.

6.3.4 Tracing Program Execution

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

To cancel a tracepoint, enter the debugger CANCEL TRACE command, specifying the program location or event exactly as you did when setting the breakpoint. The CANCEL TRACE/ALL command cancels all tracepoints.

NOTE

If the symbol on which you want to set a tracepoint is in a scope that is not visible to the debugger, you may have to set its module. See Section 6.5.2.4 for more information on the debugger SET MODULE command.

If you set a tracepoint at a location currently used as a breakpoint, the breakpoint is canceled in favor of the tracepoint. See Section 6.3.3 for more information on breakpoints.

As with breakpoints, every time a tracepoint is reached, the debugger enters a message and displays the source line. However, at tracepoints, the program continues executing, and the debugger prompt is not displayed. For example:

```
DBG> SET TRACE SUM
DBG> GO
    .
    .
    .
trace at routine MAIN.SUM
     1: function SUM (X: INTEGER) return INTEGER is
    .
    .
    .
```

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

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

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

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

```
DBG> SET TRACE/SILENT %LINE 11 DO (EXAMINE I,J,K)
DBG> GO
    .
    .
    .
MAIN.I: 7
MAIN.J: 28
MAIN.K: 176906
    .
    .
    .
```

6.3.5 Monitoring Changes in Variables

The debugger SET WATCH command allows you to set *watchpoints* that will be monitored continuously as your program executes. With high-level languages like Ada, you typically set watchpoints on variables that have been declared in your program (you can also set watchpoints on arbitrary program locations). If the program modifies the value of a watched variable, the debugger suspends execution and displays the old and new values.

To set a watchpoint on a variable, specify the variable's name with the SET WATCH command.

NOTE

If the symbol on which you want to set a watchpoint is in a scope that is not visible to the debugger, you may have to set its module. See Section 6.5.2.4 for more information on the debugger SET MODULE command.

For example, the following command sets a watchpoint on the variable TOTAL:

```
DBG> SET WATCH TOTAL
```

Subsequently, every time the program modifies the value of TOTAL, the watchpoint is triggered, and the debugger suspends execution.

NOTE

The use of watchpoints may increase the CPU time required by your program, causing a large increase in elapsed time. This time increase does not indicate an error in your program.

Like the SET BREAK and SET TRACE commands, the SET WATCH command accepts optional DO and WHEN clauses. Also, you use the SHOW WATCH and CANCEL WATCH commands to display and cancel the currently active watchpoints.

The following example shows the effect on program execution when your program modifies the contents of a watched variable:

```
DBG> SET WATCH TOTAL
DBG> GO
   .
   .
   .
watch of SUM.TOTAL at SUM.LOOP$5.%LINE 6+6
     6:       TOTAL := TOTAL + I;
   old value: 0
   new value: 1
break at SUM.LOOP$5.%LINE 7
     7:    end loop;
DBG>
```

In this example, a watchpoint is set on the variable TOTAL, and the GO command is entered to start execution. When the value of TOTAL changes, execution is suspended. The debugger reports the event ("watch of . . . ") and identifies where TOTAL changed (line 6) and the associated source line. The debugger then displays the old and new values and reports

that execution has been suspended at the start of the next line (7). (The debugger reports "break at . . . ", but this is the effect of the watchpoint, not a breakpoint.) Finally, the debugger prompts for another command.

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

This general technique for setting watchpoints always applies to static variables. A static variable is associated with the same virtual memory location throughout program execution. In VAX Ada, only some variables that are declared in library packages are statically allocated.

A variable that is allocated on the stack or in a register—a nonstatic variable—exists only when its defining routine (subprogram or task) is active (on the call stack). If you try to set a watchpoint on a nonstatic variable when its defining routine is not active, the debugger issues a warning:

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'MAIN.AVERAGE.Y' is not activ
```

A convenient technique for setting a watchpoint on a nonstatic variable is to set a breakpoint on the defining routine, specifying a DO clause to set the watchpoint whenever execution reaches the breakpoint. In the following example, a watchpoint is set on the nonstatic variable Y in routine AVERAGE:

```
DBG> SET BREAK AVERAGE DO (SET WATCH Y)
DBG> GO
   .
   .
   .
break at routine MAIN.AVERAGE
     4:    function AVERAGE (Y: INTEGER) return INTEGER is
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
DBG> SHOW WATCH
watchpoint of MAIN.AVERAGE.Y [tracing every instruction]
DBG>
```

The debugger monitors nonstatic watchpoints by tracing every instruction. Because this slows execution speed compared to monitoring static watchpoints, the debugger lets you know when it is monitoring nonstatic watchpoints.

When execution eventually returns to the calling routine, the nonstatic variable is no longer active, so the debugger automatically cancels the watchpoint and enters a message to that effect.

Note that if you specify the /OPTIMIZE qualifier (the default) when compiling your program, the compiler may remove certain variables in the program. If you try to set a watchpoint on one of these variables, the debugger issues the following warning:

```
%DEBUG-W-UNALLOCATED, entity symbol was not allocated in memory
                      (was optimized away)
```

For this reason, you should compile your program with the /NOOPTIMIZE (or /OPTIMIZE=NONE) qualifier when you prepare for debugging.

6.3.6 Debugging Ada Library Packages

When an Ada main program (or a non-Ada main program that calls Ada code) is executed, initialization code is executed for the Ada run-time library and elaboration code for all library units that the program depends on. The elaboration code causes the library units to be elaborated in appropriate order before the main program is executed. Library specifications, bodies, and some of their subunits are also elaborated by this process.

The elaboration of library packages accomplishes the following operations:

- Causes package declarations to take effect
- Initializes any variables whose declaration includes initialization code
- Executes any sequence of statements that appear between the **begin** and **end** statements of package bodies

When you run a VAX Ada main program under debugger control, execution is suspended initially before the initialization code is executed and before the elaboration of library units. For example:

```
$ RUN HOTEL

         VAX DEBUG Version V5.0-00

%DEBUG-I-INITIAL, language is ADA, module set to HOTEL
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
DBG>
```

At that point, before typing GO to get to the start of the main program, you can step through and examine parts of the library packages by setting breakpoints at the package specifications or bodies you are interested in. You then use the GO command to get to the start of each package. (See Section 6.3.1.1 for more information on the GO command.)

You set a breakpoint (or a tracepoint) on a package body by specifying the package unit name in a debugger SET BREAK (or SET TRACE) command. You set a breakpoint (or a tracepoint) on a package specification by specifying the package unit name followed by a trailing underscore character (_).

NOTE

Even if you have set a breakpoint on a package body, the break will not occur if the debugger module for that body is not set. If the module is not set, the break will occur at the package specification. This effect occurs because the debugger automatically sets modules for the specifications of packages named in **with** clauses; it does not automatically set modules for the associated package bodies.

Also, to set a breakpoint on a subprogram declared in a package specification, you must set the module for the package body.

See Section 6.5.2 for more information on module setting.

For example, the following command sets breakpoints on both the specification and body of the package RESERVATIONS:

```
DBG> SET BREAK RESERVATIONS_,RESERVATIONS
```

Note that the compiler generates unique names for subprograms declared in library packages that are or could be overloaded names. The debugger uses these unique names in its output, and requires them in commands where the names would otherwise be ambiguous. See Section 6.5.4 for more information on resolving overloaded names and symbols.

6.3.7 Monitoring Ada Exceptions

The debugger recognizes three kinds of exceptions:

- A user-defined exception—an exception declared with the Ada reserved word **exception** in an Ada compilation unit
- An Ada predefined exception, such as PROGRAM_ERROR or CONSTRAINT_ERROR
- Any other (non-Ada) exception or VMS condition

The debugger provides a number of methods for monitoring exceptions. These methods are described in the following sections.

6.3.7.1 Monitoring Any Exception

You can use the debugger commands SET BREAK/EXCEPTION and SET TRACE/EXCEPTION to cause a breakpoint or tracepoint to occur on any exception or VMS condition.

NOTE

The SET BREAK/EXCEPTION command will cause breakpoints to occur at certain VMS conditions that are signaled internally within the VAX Ada run-time library. These conditions are an implementation mechanism—they do not represent program failures, and they cannot be handled by Ada exception handlers. If these conditions appear while you are debugging your program, you may want to consider selecting the kind of exceptions that you set breakpoints on (see Sections 6.3.7.2 and 6.3.7.3).

The following example shows a tracepoint occurring for an Ada CONSTRAINT_ERROR exception as the result of a SET TRACE/EXCEPTION debugger command:

```
DBG> SET TRACE/EXCEPTION
DBG> GO
    .
    .
    .
%ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
-ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000A7C
trace on exception preceding ADA$RAISE\ADA$RAISE_CONDITION.%LINE 333+12
    .
    .
    .
```

The next example uses the SET BREAK/EXCEPTION command and the SHOW CALLS command. The SET BREAK/EXCEPTION command causes a breakpoint to occur for any exception raised; the SHOW CALLS command displays a traceback of the calls leading to the subprogram where the exception occurred or to which the exception was raised. For example:

```
DBG> SET BREAK/EXCEPTION DO (SHOW CALLS)
DBG> GO
    .
    .
    .
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=000008AF,
PSL=03C000A2 break on exception preceding SYSTEM_OPS.DIVIDE.%LINE 17+6
     17:      return X/Y;
 module name     routine name                    line       rel PC    abs PC
*SYSTEM_OPS      DIVIDE                            17      00000015  000008AF
*PROCESSOR       PROCESSOR                         19      000000AE  00000BAD
*ADA$ELAB_PROCESSOR
                 ADA$ELAB_PROCESSOR                        00000009  00000809
                 LIB$INITIALIZE                            00000054  00000C36
 SHARE$ADARTL                                              00000000  000398BE
*ADA$ELAB_PROCESSOR
                 ADA$ELAB_PROCESSOR                        0000001B  0000081B
                 LIB$INITIALIZE                            0000002F  00000C21
DBG>
```

In this example, the VAX condition SS$_INTDIV is raised at line 17 of the subprogram DIVIDE in the package SYSTEM_OPS. The example shows an important effect: some VAX conditions (such as $SS_INTDIV) are treated as being equivalent to some Ada predefined exceptions. See the *VAX Ada Run-Time Reference Manual* for a list of these conditions and exceptions.

The matching of a VAX condition and an Ada predefined exception is performed by the condition handler provided by VAX Ada for any frame that includes an exception part. Therefore, when an exception breakpoint or tracepoint is triggered by a VAX condition that has an equivalent Ada exception name, the message displays *only* the system condition code name, and not the name of the corresponding Ada exception.

6.3.7.2 Monitoring Specific Exceptions

Whenever an exception is raised, the debugger sets the symbols identified in Table 6–1. You can use these exception symbols to qualify your exception breakpoints or tracepoints so that they trigger only on certain exceptions.

For example, the following command sets a breakpoint only when a CONSTRAINT_ERROR exception is raised:

```
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NAME = "CONSTRAINT_ERRO")
DBG>
```

You can use the debugger EVALUATE command with any of these symbols to obtain information from the debugger. For example, the following command causes the value of the exception to be displayed when it is traced:

```
DBG> SET TRACE/EXCEPTION DO (EVALUATE %EXC_NUM)
DBG> GO
    .
    .
    .
%ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
-ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000BCC
trace on exception preceding SHARE$ADARTL+69323
3244836
    .
    .
    .
```

Table 6–1: Debugger Exception Symbols

Exception Symbol	Contents
%EXC_FACILITY	A string that names the facility that issued the exception. The facility name for Ada predefined exceptions and user-defined exceptions is ADA.
%EXC_NAME	An uppercase string that names the exception. If the exception raised is an Ada predefined exception, its name is truncated if it exceeds 15 characters. For example, CONSTRAINT_ERROR is truncated to CONSTRAINT_ERRO. If the exception is a user-defined exception, %EXC_NAME contains the string "EXCEPTION", and the name of the user-defined exception is contained in %ADAEXC_NAME.
%ADAEXC_NAME	If the exception raised is user-defined, %ADAEXC_NAME contains a string that names the exception, and %EXC_NAME contains the string "EXCEPTION". If the exception is not user-defined, %ADAEXC_NAME contains a null string, and the name of the exception is contained in %EXC_NAME.
%EXC_NUM	The number of the exception.
%EXC_SEVERITY	A string that gives the exception severity level (F, E, W, I, S, or ?).

6.3.7.3 Monitoring Handled Exceptions and Exception Handlers

You can use the debugger SET BREAK/EVENT=event-name and SET TRACE/EVENT=event-name commands to set breakpoints or tracepoints on exceptions that are about to be handled by Ada exception handlers. These commands allow you to observe the execution of each Ada exception handler that gains control.

The complete syntax for these commands is as follows:

```
SET BREAK/EVENT=event-name [WHEN(lang-expr)] [DO(command[;...])]
SET TRACE/EVENT=event-name [WHEN(lang-expr)] [DO(command[;...])]
```

You can specify two event names—HANDLED and HANDLED_OTHERS—as defined in Table 6–2.

Table 6–2: Exception-Related VAX Ada Event Names

Event-name	Description
HANDLED	Triggers when an exception is about to be handled in some Ada exception handler, including an **others** handler.
HANDLED_OTHERS	Triggers only when an exception is about to be handled in an **others** Ada exception handler.

For example, the following command causes a breakpoint to occur whenever an exception is about to be handled by an Ada exception handler:

```
DBG> SET BREAK/EVENT=HANDLED
```

When the breakpoint occurs, the debugger identifies the exception that is about to be handled and the exception handler that is about to be executed. You can then use that information to set a breakpoint on a particular handler, or you can type the GO command, and see which Ada handler next attempts to handle the exception. For example:

```
DBG> GO
    .
    .
    .
break on Ada event HANDLED
  task %TASK 1 is about to handle an exception
  The Ada exception handler is at: PROCESSOR.%LINE 21
    %ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
    -ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000A7C
DBG> SET BREAK PROCESSOR.%LINE 21
```

In this example, the exception CONSTRAINT_ERROR is about to be handled by an exception handler at line 21 of procedure PROCESSOR. The SET BREAK command sets a specific breakpoint on the handler at line 21.

6.4 Examining and Manipulating Data

The following sections explain how to use the debugger EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables and to evaluate expressions. They also note restrictions on the use of these commands with VAX Ada programs.

When examining and manipulating data, note the following considerations:

- You must have compiled and linked your program with the /DEBUG qualifier in effect; otherwise, the debugger will not have the information it needs about the variables in the executable image. You should also have compiled your program with the /NOOPTIMIZE (or /OPTIMIZE=NONE) qualifier (see Section 6.2.1).
- If the symbol that you want to examine or manipulate is in a scope that is not visible to the debugger, you may have to set its module. See Section 6.5.2.4 for more information on the debugger SET MODULE command.
- Before you can examine or deposit into a nonstatic variable (any variable not declared in a library package), its defining subprogram, task, and so on, must be active (must be on the call stack). In other words, program execution must be suspended somewhere within the defining routine.
- Before you can examine, deposit, or evaluate an Ada subprogram formal parameter or an Ada variable, the parameter or variable must be elaborated. In other words, you should step or otherwise move control past the parameter or variable's declaration. The value contained in any variable or formal parameter whose declaration has not been elaborated may be invalid.

See Section 6.4.5 for detailed information on debugger support of Ada constructs with the EXAMINE, DEPOSIT, and EVALUATE commands.

6.4.1 Displaying the Values of Variables

To display the current value of a variable, use the debugger EXAMINE command. The EXAMINE command has the following form:

```
EXAMINE variable-name
```

The debugger recognizes the compiler-generated data type of the specified variable and retrieves and formats the data accordingly.

You can use the EXAMINE command with any kind of address expression, not just a variable name, to display the contents of a program location. The debugger associates certain default data types with untyped locations. You can override the defaults for typed and untyped locations if you want the data to be interpreted and displayed in some other data format.

You cannot use the EXAMINE command to determine the values of Ada attributes (for example, EXAMINE MONTH'LAST) or named numbers. You must use the EVALUATE command instead. See Section 6.4.4 for more information and for a comparison of the EXAMINE and EVALUATE commands.

The following examples show some uses of the EXAMINE command; additional examples appear in Section 6.4.7.

Examine three integer variables:

```
DBG> EXAMINE WIDTH, LENGTH, AREA
EXAMPLE.WIDTH:  4
EXAMPLE.LENGTH: 7
EXAMPLE.AREA:   28
DBG>
```

Examine a two-dimensional array of integers (three per dimension):

```
DBG> EXAMINE INTEGER_ARRAY
EXAMPLE.INTEGER_ARRAY
    (1,1):      27
    (1,2):      31
    (1,3):      12
    (2,1):      15
    (2,2):      22
    (2,3):      18
DBG>
```

To determine the storage representation of a variable, you can use the SHOW SYMBOL/TYPE command. For example:

```
DBG> SHOW SYMBOL/TYPE AREA
data EXAMPLE.AREA
    atomic type, longword integer, size: 4 bytes
DBG>
```

6.4.2 Changing the Values of Variables

To change the value of a variable, use the debugger DEPOSIT command. The DEPOSIT command has the following form:

```
DEPOSIT variable-name := value
```

The DEPOSIT command is like an assignment statement in VAX Ada. In VAX Ada, you can assign a value to an object of a scalar type, an access type, or to an individual component of an object of a composite type (record or array). Except for string literals, the debugger cannot evaluate expressions of array or record types. Also, the value to be assigned must be an Ada language expression. Debugger support for Ada language expressions is identified in Section 6.4.5.2.

In the following examples, the DEPOSIT command assigns new values to different variables. The debugger checks that the value assigned is consistent with the data type and constraints of the variable. Additional examples appear in Section 6.4.7.

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

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

Note that if a string constant has fewer characters than the number specified in the string declaration, the debugger pads the remaining characters with blanks from the right. If a string constant has more characters than the number specified in the string declaration, the debugger truncates the extra characters from the right and issues the following message:

```
%DEBUG-I-ISTRTRU, string truncated at or near DEPOSIT
```

Deposit an integer expression:

```
DBG> DEPOSIT WIDTH := CURRENTWIDTH + 10
```

Note that if you deposit an out-of-range integer value, the debugger issues an informational message and truncates the high-order bits.

Deposit element 2 of an array of characters (you cannot deposit an entire array aggregate with a single DEPOSIT command, only an element):

```
DBG> DEPOSIT C_12_ARRAY(2) := 'K'
```

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

6.4.3 Current, Previous, and Next Locations

When using the debugger EXAMINE and DEPOSIT commands, you can use three special debugger operators to refer to the current, previous, and next data locations. The operators are the period (.), the circumflex (^), and the RETURN key.

The period (.) denotes the current location—that is, the location most recently referenced by an EXAMINE or DEPOSIT command. For example:

```
DBG> EXAMINE WIDTH
EXAMPLE.WIDTH: 15
DBG> DEPOSIT . := 12
DBG> EXAMINE .
EXAMPLE.WIDTH: 12
DBG>
```

Similarly, the circumflex (^) denotes the previous location, and the RETURN key denotes the next location. These operators are useful for referring to consecutive indexed components of an array. For example, if INTEGER_ARRAY is a two-dimensional array of integers, then you can examine it as follows:

```
DBG> EXAMINE INTEGER_ARRAY(1,2)
EXAMPLE.INTEGER_ARRAY(1,2):     31
DBG> EXAMINE ^
EXAMPLE.INTEGER_ARRAY(1,1):     27
DBG> EXAMINE
EXAMPLE.INTEGER_ARRAY(1,2):     31
```

6.4.4 Evaluating Expressions

To evaluate a language expression, use the debugger EVALUATE command. You must use the EVALUATE command to determine the value of an Ada attribute or named number.

NOTE

The debugger is not as strongly typed or precise as the Ada language. Thus, the evaluation of an expression by the EVALUATE command may differ from the result that would be calculated by Ada-generated code and obtained with the EXAMINE command.

The EVALUATE command has the following form:

```
EVALUATE language-expression
```

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

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

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

```
DBG> DEPOSIT WIDTH := 45
DBG> EVALUATE WIDTH
45
DBG> EXAMINE WIDTH
EXAMPLE.WIDTH:  45
```

The following example shows an important difference between the EVALUATE and EXAMINE commands:

```
DBG> EVALUATE WIDTH + 7
52
DBG> EXAMINE WIDTH + 7
EXAMPLE.WIDTH+7:        -310853632
```

With the EVALUATE command, WIDTH + 7 is interpreted as a language expression, which evaluates to 45 + 7, or 52. With the EXAMINE command, WIDTH + 7 is interpreted as an address expression: 7 bytes are added to the address of WIDTH, and whatever value is in the resulting address is reported (in this instance, –310853632).

See Section 6.4.7 for additional examples of the EVALUATE and EXAMINE commands.

6.4.5 Debugger Support for VAX Ada Data

In general, the debugger supports the data types and operators of VAX Ada. However, there are certain language-specific limitations or other differences. This section presents those limitations or differences, roughly following the organization of Chapters 2 and 4 of the *VAX Ada Language Reference Manual*.

For information on the supported data types and operators of any of the debugger-supported languages, type the debugger HELP LANGUAGE command at the DBG> prompt.

6.4.5.1 Ada Names

Table 6–3 presents the debugger support for Ada names. In this table, "fully supported" has the following meaning:

- When you use the debugger EXAMINE and DEPOSIT commands, you can specify a variable name exactly as you might in the program.
- When you use the debugger DEPOSIT command, the debugger performs the same type and constraint checks that the compiler and VAX Ada run-time library do.

Note that parts of names may be language expressions—for example, attributes such as FIRST or POS. This affects how you use the EXAMINE, EVALUATE, and DEPOSIT commands with such names (see the examples of enumeration types in Section 6.4.7.1).

Table 6–3: Debugger Support for Ada Names

Kind of Name	Debugger Support
Lexical elements	Full support for Ada rules for the syntax of identifiers.
	Function designators that are operator symbols (for example, + and *) rather than identifiers must be prefixed with %NAME. Also, the operator symbol must be enclosed in quotation marks.
	Full support for Ada rules for numeric literals, character literals, string literals, and reserved words.
	The debugger accepts signed integer literals in the range –2147483648 to 2147483647.
	Depending on context, the debugger interprets floating-point types as F_floating, D_floating, G_floating, or H_floating.

(continued on next page)

Table 6–3 (Cont.): Debugger Support for Ada Names

Kind of Name	Debugger Support
Function calls	You cannot use function calls with the EXAMINE, EVALUATE, or DEPOSIT commands. For example, if PRODUCT is a function that multiplies two integers, you cannot enter the command EVALUATE PRODUCT(3,5). If your program assigns the returned value of a function to a variable, you can then examine the value of that variable. The only commands that cause the program to execute are GO, STEP, CALL, and EXIT (EXIT executes exit handlers). See Section 6.6.5 for summary information on the CALL command.
Indexed components	Full support.
Slices	You can examine and evaluate an entire slice or an indexed component of a slice. You can deposit only to an indexed component of a slice. You cannot deposit an entire slice.
Selected components	Full support, including use of the keyword **all** in **.all**.
Attributes	Only the predefined attributes in Table 6–4 are supported. Note that the debugger SHOW SYMBOL/TYPE command provides the same information that is provided by the P'FIRST, P'LAST, P'LENGTH, P'SIZE, and P'CONSTRAINED attributes.
Literals	Full support, including the keyword **null**.
Aggregates	You can examine the entire record and array objects with the EXAMINE command. You can deposit a value in a component of an array or record. You cannot use the DEPOSIT command with aggregates, except to deposit character string values.

Table 6–4: Debugger Support for Ada Predefined Attributes

Attribute	Debugger Support
P'CONSTRAINED	For a prefix P that denotes a record object with discriminants. The value of P'CONSTRAINED reflects the current state of P (constrained or unconstrained).
P'FIRST	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the lower bound of P.
P'FIRST	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the lower bound of the first index range.
P'FIRST(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the lower bound of the Nth index range.
P'LAST	For a prefix P that denotes an enumeration type, or a subtype of an enumeration type. Yields the upper bound of P.
P'LAST	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the upper bound of the first index range.
P'LAST(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the upper bound of the Nth index range.
P'LENGTH	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the number of values of the first index range (zero for a null range).
P'LENGTH(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the number of values of the Nth index range (zero for a null range).
P'POS(X)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the position number of the value X. The first position is 0.
P'PRED(X)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the value of type P, which has a position number one less than that of X.

(continued on next page)

Table 6–4 (Cont.): Debugger Support for Ada Predefined Attributes

Attribute	Debugger Support
P'SIZE	For a prefix P that denotes an object. Yields the number of bits allocated to hold the object.
P'SUCC(X)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the value of type P, which has a position number one more than that of X.
P'VAL(N)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the value of type P, which has the position number N. The first position is 0.

6.4.5.2 Ada Language Expressions

Table 6–5 presents the debugger support for Ada language expressions. You use language expressions with the debugger EVALUATE and DEPOSIT commands. For example:

```
DBG> EVALUATE PART_NUMBER(CURRENTWIDTH+1)
'9'
DBG> DEPOSIT WIDTH := 56
DBG>
```

Language expressions can also appear in a debugger WHEN or DO clause. For example:

```
DBG> SET BREAK/INSTRUCTION WHEN (A /= 0)
DBG>
```

Table 6–5: Debugger Support for Ada Language Expressions

Kind of Expression	Debugger Support
Operators	Table 6–6 indicates the debugger support for the predefined Ada operators. The debugger does *not* support the following: • Operations on entire arrays or records • The short-circuit control forms: **and then, or else** • The membership tests: **in, not in** • User-defined operators

(continued on next page)

Table 6–5 (Cont.): Debugger Support for Ada Language Expressions

Kind of Expression	Debugger Support
Type conversions	No support for any of the *explicit* type conversions specified in Ada. However, the debugger performs certain *implicit* type conversions between numeric types during the evaluation of expressions. The debugger converts lower-precision types to higher-precision types before evaluating expressions involving types of different precision: • If integer and floating-point types are mixed, the integer type is converted to floating-point type. • If integer and fixed-point types are mixed, the integer type is converted to fixed-point type. • If integer types of different sizes are mixed (for example, byte-integer and word-integer), the one with the smaller size is converted to the larger size. In general, the debugger allows more implicit type conversion than the Ada language. In addition, the hardware type used for a debugger calculation may differ from that chosen by the compiler.
Subtypes	Full support. Note that the debugger denotes subtypes and types that have range constraints as "subrange" types.
Qualified expressions	Supported as required to resolve overloaded enumeration literals (literals that have the same identifier but belong to different enumeration types). The debugger does not support qualified expressions for any other purpose.
Allocators	No support for any operations with allocators.
Universal expressions	No support.

Table 6–6: Debugger Support for Ada Predefined Operators

Operator	Operation	Debugger Support
and	Conjunction	Full support, except for bit arrays
or	Inclusive disjunction	Full support, except for bit arrays
xor	Exclusive disjunction	Full support, except for bit arrays

(continued on next page)

Table 6–6 (Cont.): Debugger Support for Ada Predefined Operators

Operator	Operation	Debugger Support
=	Equality	Only scalar and string types
/=	Inequality	Only scalar and string types
<	Less than	Only scalar and string types
<=	Less than or equal	Only scalar and string types
>	Greater than	Only scalar and string types
>=	Greater than or equal	Only scalar and string types
+	Addition	Full support
–	Subtraction	Full support
&	Catenation	Only string types
+	Identity	Full support
–	Negation	Full support
*	Multiplication	Full support
/	Division	Full support
mod	Modulus	Full support
rem	Remainder	Full support
abs	Absolute value	Full support
not	Logical negation	Full support
**	Exponentiation	Full support

6.4.6 Special EXAMINE, DEPOSIT, and EVALUATE Options

The following sections summarize the use of additional options for the debugger EXAMINE, DEPOSIT, and EVALUATE commands.

6.4.6.1 Specifying Data Type and Radix

When you examine an Ada variable or deposit a value into one, you do not need to specify the data type of the variable, unless you want to alter the way in which data is displayed or deposited. By default, the debugger uses the declared data type of a variable (including constraints) and the decimal radix for numeric values.

You can override the default data type or radix by using a command qualifier, such as /ASCII, /BINARY, /BYTE, /WORD, /LONG, /G_FLOAT, /DECIMAL, /HEXADECIMAL, and so on. The command qualifier causes the debugger to take the current type representation or radix and convert it to the one required by the qualifier. For example, consider the following declaration:

```
CHAR: CHARACTER := 'B';
```

You can examine the value of CHAR as follows:

```
DBG> EXAMINE CHAR
EXAMPLE.CHAR:   'B'
DBG>
```

You can then determine the ASCII decimal value of CHAR as follows:

```
DBG> EXAMINE/DECIMAL CHAR
EXAMPLE.CHAR:   66
DBG>
```

When an EXAMINE or DEPOSIT command specifies a location that has no associated data type, the debugger uses its default type to determine how many bytes to display or modify. The debugger also uses the default type whenever an EXAMINE or DEPOSIT command specifies a virtual address or an address expression more complicated than a single symbolic reference.

For VAX Ada, the debugger's default type for data is LONG integer; its default type for line number references is INSTRUCTION. You can change either default type with the SET TYPE command. You can use the SET TYPE/OVERRIDE command to control the operation of all succeeding EXAMINE and DEPOSIT commands that do not explicitly specify a type. For example:

```
DBG> SET TYPE/OVERRIDE BYTE
DBG>
```

After this command is entered, an unqualified EXAMINE or DEPOSIT command displays or modifies only the first byte of any variable's storage. To restore the normal interpretation of data types, enter the CANCEL TYPE/OVERRIDE command.

6.4.6.2 Obtaining Virtual Addresses

By default, the debugger displays virtual addresses in decimal radix. However, VMS virtual addresses are expressed in hexadecimal radix in various contexts, such as linker maps or the debugger's SHOW CALLS display. This section describes how to use special EXAMINE and EVALUATE command qualifiers to determine virtual addresses and specify the radix.

You can determine the virtual address of a line number, module, routine, or variable by using the EVALUATE/ADDRESS command. By default, the address is displayed in decimal radix, but if you further specify the /HEXADECIMAL qualifier, the debugger displays the virtual address in hexadecimal radix. For example:

```
DBG> EVALUATE/ADDRESS/HEXADECIMAL %LINE 20
00000CF5
DBG> EVALUATE/ADDRESS/HEXADECIMAL EXAMPLE
00000DEA
DBG>
```

If a variable is stored in a register instead of in virtual memory, the EVALUATE/ADDRESS command returns the name of the register.

When you examine an access object, you obtain the location in virtual memory of the designated object (see Section 6.4.7.4). The value is provided in decimal radix by default, but you can use the /HEXADECIMAL qualifier to convert it to hexadecimal radix. For example:

```
DBG> EXAMINE/HEXADECIMAL A
EXAMPLE.A:      00A2000
DBG>
```

You can use the SET RADIX command to alter the radix for numeric data. For example, the SET RADIX HEXADECIMAL command causes all numeric data to be interpreted and displayed by the debugger in hexadecimal radix. To restore the decimal radix, use the SET RADIX DECIMAL command.

6.4.7 Ada Data Types—Debugging Examples

The following sections provide examples showing how to use the debugger EXAMINE, EVALUATE, and DEPOSIT commands with selected Ada data types. These sections are organized according to data type, as in the *VAX Ada Language Reference Manual*. The examples include the use of Ada names and language expressions.

Several examples show the kinds of checks the debugger performs and the messages entered when there is an error. In particular, the debugger checks for the following:

- Values that are out of bounds, for any discrete type
- Index values that are out of bounds, for any constrained array type
- Type conflicts—for example, when you use the DEPOSIT command

6.4.7.1 Scalar Types

The examples of scalar types include enumeration types, integer types, and floating-point types.

Enumeration Types

Consider the following declarations:

```
type DAY is
   (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY);
MY_DAY : DAY;
```

You can determine the debugger's storage representation for type DAY as follows:

```
DBG>  SHOW SYMBOL/TYPE DAY
type EXAMPLE.DAY
    enumeration type (DAY, 7 elements), size: 1 byte
DBG>
```

The following examples show the use of attributes with enumeration types. Note that you cannot use the EXAMINE command to determine the value of attributes, because attributes are not variable names. You must use the EVALUATE command instead. For the same reason, attributes can appear only on the right of the := operator in a DEPOSIT command.

```
DBG>  EVALUATE DAY'FIRST
MONDAY
DBG>  EVALUATE DAY'LAST
SUNDAY
DBG>  EVALUATE DAY'POS(WEDNESDAY)
2
DBG>  EVALUATE DAY'VAL(4)
FRIDAY
DBG>  DEPOSIT MY_DAY := TUESDAY
DBG>  EVALUATE DAY'SUCC(MY_DAY)
WEDNESDAY
DBG>  DEPOSIT . := DAY'PRED(MY_DAY)
DBG>  EXAMINE .
EXAMPLE.MY_DAY: MONDAY
DBG>  EVALUATE DAY'PRED(MY_DAY)
%DEBUG-W-ILLENUMVAL, enumeration value out of legal range
DBG>
```

Consider the following declarations:

```
type MASK is (DEC,FIX,EXP);
type CODE is (FIX,CLA,DEC);
MY_MASK : MASK;
MY_CODE : CODE;
```

In the following example, the qualified expression CODE'(FIX) resolves the overloaded enumeration literal FIX, which belongs to both type CODE and type MASK:

```
DBG> DEPOSIT MY_CODE := FIX
%DEBUG-W-NOUNIQUE, symbol 'FIX' is not unique
DBG> SHOW SYMBOL/TYPE FIX
data EXAMPLE.FIX
    enumeration type (CODE, 3 elements), size: 1 byte
data EXAMPLE.FIX
    enumeration type (MASK, 3 elements), size: 1 byte
DBG> DEPOSIT MY_CODE := CODE'(FIX)
DBG> EXAMINE MY_CODE
EXAMPLE.MY_CODE:        FIX
DBG>
```

Integer Types

Consider the following declarations:

```
type INDEX is range 1 .. 700;
subtype SMALL_INDEX is INDEX range 1 .. 20;
MY_INDEX : INDEX;
MY_SMALL_INDEX : SMALL_INDEX;
```

You can determine the debugger's storage representations for types INDEX and SMALL_INDEX as follows:

```
DBG> SHOW SYMBOL/TYPE INDEX, SMALL_INDEX
type EXAMPLE.INDEX
    subrange type, size: 2 bytes, range: 1..700
        parent type: atomic type, word integer, size: 2 bytes
type EXAMPLE.SMALL_INDEX
    subrange type, size: 2 bytes, range: 1..20
        parent type: atomic type, word integer, size: 2 bytes
DBG>
```

In the next example, the value 30000 is out of bounds for the subtype MY_SMALL_INDEX:

```
DBG> DEPOSIT MY_INDEX := 50
DBG> DEPOSIT MY_SMALL_INDEX := 30000
%DEBUG-I-IVALOUTBNDS, value assigned is out of bounds at or
    near DEPOSIT
DBG>
```

Note that the severity level of the message in this example is I (informational), so the debugger executes the command even though it enters the message. Unlike Ada, the debugger allows the out-of-bounds assignment to be performed in this case, truncating high-order bits if the assignment value will not fit into the storage allocated. For example:

```
DBG> EXAMINE MY_SMALL_INDEX
EXAMPLE.MY_SMALL_INDEX: -25536
DBG>
```

The debugger terminates an illegal command line only when the severity level of the message is W (warning) or greater.

Real Types

Consider the following declarations:

```
type VOLUME is digits 7 range 0.0..1.0E6;
type FIXED_POINT is delta 0.25 range 0.0..25.0;
MY_VOLUME : VOLUME;
FP: FIXED_POINT;
```

You can determine the debugger's storage representation for type VOLUME as follows:

```
DBG> SHOW SYMBOL/TYPE VOLUME
type EXAMPLE.VOLUME
    subrange type, size: 8 bytes,
        range: 0.000000000000000..1000000.00000000
          parent type: atomic type, G_floating, size: 8 bytes
DBG>
```

Similarly, the debugger's representation for type FIXED_POINT is as follows:

```
DBG> SHOW SYMBOL/TYPE FIXED_POINT
type EXAMPLE.FIXED_POINT
    subrange type, size: 1 byte, range: 0.00..25.00
        parent type: binary scalar string descriptor type,
          byte integer, size: 1 byte scaled -2
DBG>
```

If you try to deposit an out-of-bounds value (in this example a negative value), you receive the following result:

```
DBG> DEPOSIT MY_VOLUME := -12.0
%DEBUG-I-IVALOUTBNDS, value assigned is out of bounds at or near -
DBG>
```

Note that if numeric types are mixed in an expression, the debugger performs a type conversion as discussed in Section 6.4.5.2. The debugger uses the hardware representation that provides at least the precision specified in the type declaration or required to evaluate an expression. For example:

```
DBG> DEPOSIT MY_VOLUME := 2.356
DBG> EXAMINE MY_VOLUME
EXAMPLE.MY_VOLUME:       2.35600000000000
DBG> EVALUATE MY_VOLUME + 3
5.35600000000000
DBG>
```

6.4.7.2 Array Types

The following examples show operations on various array types: string, nonstring, and multidimensional arrays. The examples show how to use indexed components, slices, and full arrays.

String Arrays

Consider the following declarations:

```
PART_NUMBER : STRING(1 .. 12);
PART_NUMBER := "LP-3592.6-84";
```

You can examine or deposit an entire string, or a slice, or a single component (character). The debugger displays string values in horizontal ASCII format, enclosed in quotation characters, rather than in the vertical format used for nonstring arrays. When depositing values, you must enclose the appropriate string, slice, or character value in apostrophes or quotation characters. For example:

```
DBG> EXAMINE PART_NUMBER
EXAMPLE.PART_NUMBER(1..12):      "LP-3592.6-84"
DBG> DEPOSIT PART_NUMBER := "WG-7619.3-84"
DBG> EXAMINE PART_NUMBER(4..9)
EXAMPLE.PART_NUMBER(4..9):       "7619.3"
DBG>
```

A single component (a character) is displayed within apostrophes. For example:

```
DBG> EXAMINE PART_NUMBER(2)
EXAMPLE.PART_NUMBER(2): 'G'
DBG> DEPOSIT PART_NUMBER(8) := '/'
DBG> EXAMINE PART_NUMBER
EXAMPLE.PART_NUMBER(1..12):      "WG-7619/3-84"
DBG>
```

Nonstring Arrays

Consider the following declarations, assuming that all variables have been initialized:

```
type CAR is (BUICK, FORD, HONDA, MERCEDES, PLYMOUTH);
type CAR_ARRAY is array(CAR) of INTEGER;
CAR_NUM : CAR_ARRAY;
```

You can examine an entire array, or a slice, or a single indexed component. For example:

```
DBG> EXAMINE CAR_NUM
EXAMPLE.CAR_NUM
     (BUICK):     39
     (FORD):      57
     (HONDA):     109
     (MERCEDES):  36
     (PLYMOUTH):  87
DBG> EXAMINE CAR_NUM(FORD..MERCEDES)
EXAMPLE.CAR_NUM
     (FORD):      57
     (HONDA):     109
     (MERCEDES):  36
DBG> EXAMINE CAR_NUM(FORD)
EXAMPLE.CAR_NUM(FORD):  57
DBG>
```

You can evaluate array expressions that include attributes. For example:

```
DBG> EVALUATE CAR_NUM(CAR'FIRST) + CAR_NUM(CAR'SUCC(CAR'FIRST))
96
DBG>
```

You can deposit only a single indexed component of a nonstring array at a time (you cannot deposit a nonstring array aggregate). For example:

```
DBG> DEPOSIT CAR_NUM(BUICK) := 12
DBG>
```

Multidimensional Arrays

Consider the following declarations, assuming that all variables have been initialized:

```
type CHAR_ARRAY is array(1..2, 1..3) of CHARACTER;
type FLOAT_ARRAY is array(1..2, 1..3) of FLOAT;
CH_ARRAY : CHAR_ARRAY;
F_ARRAY : FLOAT_ARRAY;
```

The operations that are allowed for multidimensional string and nonstring arrays are the same as those allowed for one-dimensional string and nonstring arrays, respectively. The debugger displays nonstring multidimensional arrays as shown in the following example:

```
DBG> EXAMINE F_ARRAY
EXAMPLE.F_ARRAY
     (1,1):       27.01000
     (1,2):       31.00000
     (1,3):       12.48000
     (2,1):       15.08000
     (2,2):       22.30000
     (2,3):       18.73000
DBG>
```

Character arrays are treated as string arrays. In the case of multidimensional arrays of characters, the debugger displays each "string" horizontally in ASCII format, as for a one-dimensional string array. For example:

```
DBG> EXAMINE CH_ARRAY
EXAMPLE.CH_ARRAY
     (1)(1:3):     "ABC"
     (2)(1:3):     "DEF"
DBG>
```

The following examples show how to deposit to individual indexed components and show the kinds of checks the debugger performs on arrays in general:

```
DBG> DEPOSIT F_ARRAY(2,1) := 3.92
DBG> DEPOSIT CH_ARRAY(1,3) := '?'
DBG>
```

If you try to deposit to an out-of-bound index value (1,4), you receive an informational warning, as follows:

```
DBG> DEPOSIT CH_ARRAY(1,4) := 'u'
%DEBUG-I-SUBOUTBND, subscript 2 is out of bounds value is 4,
    bounds are 1..3
DBG>
```

Because the warning is only informational, however, the debugger assigns the value as best it can. The result of the operation in this example is as follows:

```
DBG> EXAMINE CH_ARRAY
EXAMPLE.CH_ARRAY
   (1)(1:3):    "AB?"
   (2)(1:3):    "uEF"
DBG>
```

If you try to deposit a floating-point type into the character array, you receive an error message, as follows:

```
DBG>  DEPOSIT CH_ARRAY(2,1) := 12.77
%DEBUG-E-OPTNOTALLOW, operator 'DEPOSIT' not allowed on given
    data types
DBG>  EXAMINE CH_ARRAY
EXAMPLE.CH_ARRAY
   (1)(1:3):    "AB?"
   (2)(1:3):    "uEF"
DBG>
```

6.4.7.3 Record Types

The following examples show operations on records with nested records or arrays and records with variant parts. Each example shows how to specify both full records and selected record components.

Records with Nested Records or Arrays

Consider the following declarations, assuming that all variables have been initialized:

```
type ADDRESS is
   record
      NUMBER : INTEGER;
      STREET : STRING(1..12);
   end record;
type CHILDREN_ARRAY is array (1..3) of STRING(1..11);
type FAMILY is
   record
      FATHER_NAME, MOTHER_NAME : STRING(1..12);
      THEIR_ADDRESS : ADDRESS;                -- A nested record.
      CHILDREN_NAME : CHILDREN_ARRAY;         -- A nested array.
   end record;

THIS_FAMILY : FAMILY;
```

The following example shows how to examine and manipulate data in the record variable THIS_FAMILY:

```
DBG> EXAMINE THIS_FAMILY
EXAMPLE.THIS_FAMILY
    FATHER_NAME(1..12): "John Brown  "
    MOTHER_NAME(1..12): "Betsy Brown "
    THEIR_ADDRESS
        NUMBER: 42
        STREET(1..12):  "Walnut St.  "
    CHILDREN_NAME
        (1)(1..6):      "Christopher"
        (2)(1..6):      "Willy      "
        (3)(1..6):      "           "
DBG> DEPOSIT THIS_FAMILY.CHILDREN_NAME(3) := "Jenny      "
DBG> EXAMINE THIS_FAMILY.CHILDREN_NAME
EXAMPLE.THIS_FAMILY.CHILDREN_NAME
    (1)(1..6):      "Christopher"
    (2)(1..6):      "Willy      "
    (3)(1..6):      "Jenny      "
DBG> DEPOSIT THIS_FAMILY.THEIR_ADDRESS.STREET := "Maple St.  "
DBG> EXAMINE THIS_FAMILY.THEIR_ADDRESS
EXAMPLE.THIS_FAMILY.THEIR_ADDRESS
    NUMBER:    42
    STREET(1..12):      "Maple St.  "
DBG>
```

Records with Variant Parts

Consider the following declarations, assuming that all variables have been initialized:

```
type STATUS_TYPE is (IN_STOCK, ON_ORDER);
type MODEL_TYPE is (SEDAN, HATCHBACK, CONVERTIBLE);
type COLOR_TYPE is (WHITE, GRAY, BLUE, RED);
type CAR_RECORD(STATUS : STATUS_TYPE := IN_STOCK) is
   record
      MODEL : MODEL_TYPE;
      COLOR : COLOR_TYPE;
      case STATUS is
         when IN_STOCK =>
            SERIAL_NUMBER : STRING(1 .. 10);
         when ON_ORDER =>
            ORDER_DATE : STRING(1 .. 11);
      end case;
   end record;

ONE_CAR : CAR_RECORD := (MODEL => SEDAN, COLOR => RED,
                         STATUS => IN_STOCK,
                         SERIAL_NUMBER => "2014FZS-OH");
```

The discriminant (STATUS) has a default value of IN_STOCK. You can examine an entire record object that has been declared with the default variant value. For example:

```
DBG> EXAMINE ONE_CAR
EXAMPLE.ONE_CAR
    STATUS:      IN_STOCK
    MODEL:       SEDAN
    COLOR:       RED
    Variant Record with Discriminant Value IN_STOCK
        SERIAL_NUMBER(1..10):    "2014FZS-OH"
DBG>
```

Note that the debugger tries to examine or assign a value to a component of a variant part that is not active, but because this is an illegal action in Ada, the debugger also issues an informational message. For example:

```
DBG> DEPOSIT ONE_CAR.STATUS := ON_ORDER
DBG> EXAMINE ONE_CAR.SERIAL_NUMBER
%DEBUG-I-BADDISCVAL, incorrect value of 1 in discriminant
    field STATUS
EXAMPLE.ONE_CAR.SERIAL_NUMBER(1..10):     "2014FZS-OH"
```

6.4.7.4 Access Types

The following examples show operations on objects of access types and explain how to debug incomplete access types.

Objects of Access Types

Consider the following declarations:

```
type PERSON_RECORD;
type LIST is access PERSON_RECORD;
type PERSON_RECORD is
   record
      NAME : STRING(1..10);
      AGE  : 1..100;
      NEXT : LIST;
   end record;

A, B, : LIST := new PERSON_RECORD;
```

A and B are access objects of type LIST that have been created by the allocator **new** PERSON_RECORD. The debugger does not support allocators, so you cannot create new access objects with the debugger.

Figure 6–2: Access Objects in Virtual Memory

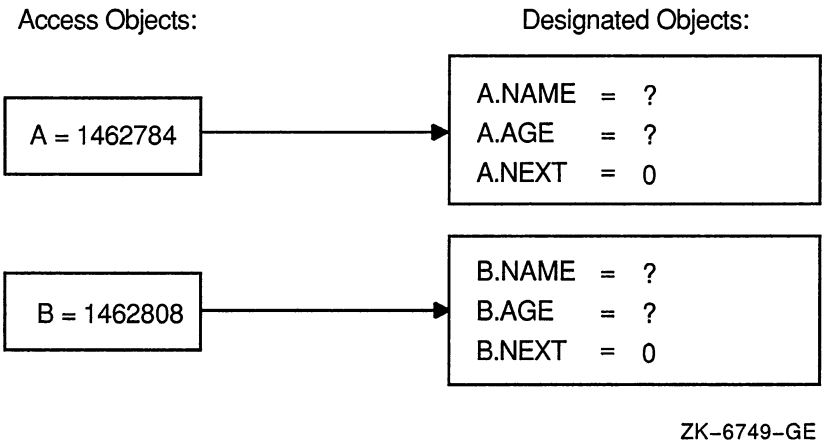


Figure 6–2 shows how the access objects, A and B, and the objects they designate (point to) exist in virtual memory.

When A and B were created, their NEXT component, was initialized to the value **null**. A and B contain the virtual memory locations of the objects they designate (1462784 and 1462808, respectively, in this case). When you specify the name of an access object with the EXAMINE command, the debugger displays the memory location of the object it designates. For example:

```
DBG> EXAMINE A, B
EXAMPLE.A:       1462784
EXAMPLE.B:       1462808
DBG>
```

By default, the debugger displays memory locations in decimal radix. The EXAMINE command has several qualifiers that allow you to display numeric values in different radixes. For example, you would display the preceding memory locations in hexadecimal radix as follows:

```
DBG> EXAMINE/HEXADECIMAL A, B
EXAMPLE.A: 00165200
EXAMPLE.B: 00165218
DBG>
```

The following examples show how to perform the following operations:

- Examine designated objects and their components.

- Assign values to the NAME and AGE components of an object of type LIST.
- Create a linked list of objects of type LIST by assigning values to the NEXT component of an object.

To examine the value of a designated object, you must use selected component notation, specifying .ALL. For example, consider that A has been initialized as follows:

```
A.NAME := "John Doe  ";
A.AGE  := 6;
A.NEXT := B;
```

Then, to examine the value of the object designated by A you would enter the following command:

```
DBG> EXAMINE A.ALL
EXAMPLE.A.ALL
    NAME(1..10):        "John Doe  "
    AGE :       6
    NEXT:       1462808
DBG>
```

To examine one component of a designated object, you can omit .ALL from the selected component syntax. For example:

```
DBG> EXAMINE A.NAME
EXAMPLE.A.ALL.NAME(1..10):     "John Doe  "
DBG>
```

Alternatively, you could have specified the following:

```
DBG> EXAMINE A.ALL.NAME
EXAMPLE.A.ALL.NAME(1..10):     "John Doe  "
DBG>
```

Now assume that the object designated by B and by A.NEXT has been initialized to the following component values:

```
B.NAME := "Sam Spade  "
B.AGE  := 3
```

The following example shows how to examine the components of the object designated by A.NEXT:

```
DBG> EXAMINE A.NEXT.ALL
EXAMPLE.A.ALL.NEXT.ALL
    NAME(1..10):        "Sam Spade  "
    AGE:        3
    NEXT:       0
DBG>
```

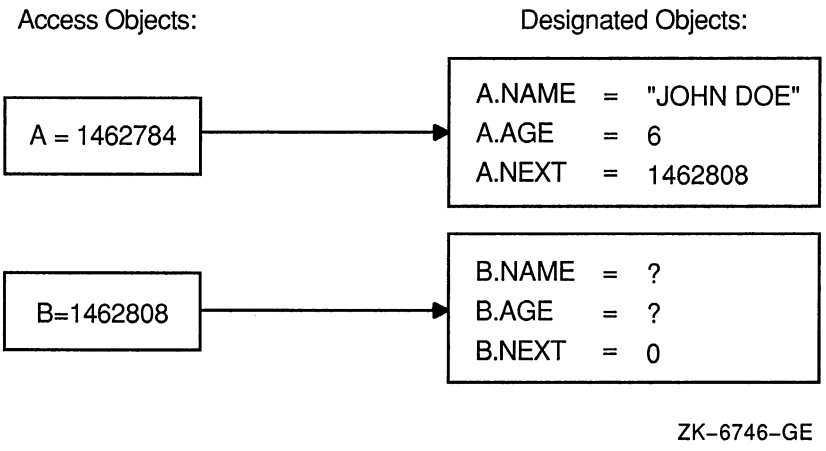
Note that you can deposit only to selected components; you cannot deposit an aggregate. For example:

```
DBG> DEPOSIT A.NAME := "John Doe  "
DBG> DEPOSIT A.AGE := 6
DBG> DEPOSIT A.NEXT := B
DBG>
```

After these commands have been executed, A.NEXT has the value contained in B, namely 1462808. Thus, A.NEXT designates the same object as B.

Figure 6–3 shows the result of these DEPOSIT assignments.

Figure 6–3: Depositing to Access Object Components



Access to Incomplete Types

Consider the following declarations:

```
package P is
   type T is private;
private
   type T_TYPE;
   type T is access T_TYPE;
end P;

package body P is
   type T_TYPE is
      record
        A: NATURAL := 5;
        B: NATURAL := 4;
      end record;

   T_REC: T_TYPE;
   T_PTR: T := new T_TYPE'(T_REC);
end P;

with P; use P;
procedure INCOMPLETE is
   VAR: T;
begin
    . . .
end INCOMPLETE;
```

The debugger does not have complete information about the type T, so you cannot manipulate the variable VAR. However, the debugger does have information about objects declared in the package body P. Thus, you can manipulate the variables T_PTR and T_REC.

Modules for package bodies are not automatically set by the debugger. To set a module, use the debugger SET MODULE command. For more information on module setting and the SET MODULE command, see Section 6.5.2

6.5 Controlling Symbol References

In general, the debugger automatically finds symbols in a manner that is consistent with Ada's scope and visibility rules. (The term *symbol* denotes names, operators, line numbers, and so on.) Thus, you can use any symbol visible at the point of execution (that is, in the PC scope) directly in debugger commands.

The debugger also allows you to reference symbols that are not visible at the point of execution. However, in such cases you may have to use some special debugger techniques to help the debugger locate or properly interpret the symbols.

These techniques and the mechanisms behind them are described in the following sections.

6.5.1 Creating Symbol Information for the Debugger

When you compile your program with the /DEBUG qualifier on any of the compilation commands (the default in VAX Ada), the compiler creates symbol records in the debug symbol table (DST) and puts them in the associated object module. *Symbols* include names of constants, variables, subprograms, packages, and so on, as well as operators and line numbers. *Symbol records* consist of symbols plus any information the debugger needs to properly interpret them as you use them in debugger commands. For example, in the case of variables, symbol records contain type, size, and constraint information; in the case of compilation units, symbol records identify specifications, bodies, subunits, and their relationships, and any additional dependences resulting from **with** and **use** clauses.

DST records contain information about all of the symbols that are defined (declared) in your program. Symbols can be either *local* or *global*. Typically, local symbols are symbols that are referenced only within the module where they are defined; global symbols are symbols that are defined in one module but referenced by others. In VAX Ada, all of the symbols you use in your program are local symbols, with the following exception: symbols that you export with export pragmas are global symbols. Global symbols are the only symbols placed in object modules when you compile units by specifying the /NODEBUG qualifier on any of the compilation commands.

When you link your program with the ACS LINK/DEBUG command, the linker copies any symbol records that are in the object files into a DST that is in the executable image. The DST also contains traceback information. Traceback information includes the names of compilation units, subprograms, and packages, and the compiler-assigned line numbers as they might appear on a listing (.LIS) file.

To facilitate symbol searches, the debugger loads symbol records from the DST into a run-time symbol table (RST), which is structured for efficient symbol lookup. During a debugging session, the debugger searches the RST (not the DST) for any symbols you specify in debugger commands. The debugger cannot recognize symbols that are not in the RST.

Section 6.5.2 explains how the debugger loads modules into the RST.

6.5.2 Module Setting

Because the RST takes up memory and loading the RST takes time, the debugger loads it dynamically, anticipating what symbols you might want to reference as your program executes during a debugging session. The loading process is called *module setting*, because all of the symbol records of a given module are loaded into the RST at one time.

A debugger module always corresponds to a compilation unit. To conserve space at run time and maximize system response, not all modules of a program are automatically set when you begin a debugging session. Rather, the debugger anticipates which modules need to be set and sets them gradually as you execute the program.

NOTE

You may need to set the modules for library package bodies yourself so that you can debug the package body or debug subprograms declared in the corresponding package specification.

See Section 6.3.6 for more information on debugging Ada library packages.

Modules are set so that you can reference any symbol that is visible in the program at the point of execution—that is, in the PC scope. Thus, if you are debugging in screen mode, you can reference any of the symbols that appear in the source code display.

To make symbols visible, the debugger performs two kinds of module setting automatically and simultaneously: *dynamic module setting* and *related module setting*. You can determine which modules have been set with the debugger SHOW MODULE command. You can explicitly control module setting with the debugger SET MODULE and CANCEL MODULE commands. The following sections describe the two kinds of module setting and the use of the various module-related commands.

6.5.2.1 Dynamic and Related Module Setting

In dynamic module setting, the debugger sets the module enclosing the PC location whenever the debugger interrupts execution (whenever the debugger prompt is displayed). Dynamic module setting makes the symbols in that module accessible to debugger commands. When setting a module dynamically, the debugger displays a message like the following:

```
%DEBUG-I-DYNMODSET, setting module X
```

Dynamic module setting makes the debugger easier to use; however, it may slow the debugger down as more symbol records are loaded into the RST. If performance becomes a problem, you can use the CANCEL MODULE command selectively, or you can turn off dynamic module setting by entering the SET MODE NODYNAMIC command.

Related module setting is an Ada-specific debugger feature. In related module setting, the debugger sets any module whose symbols should be visible within the module that is being set dynamically. In other words, when it sets a given module dynamically, the debugger also sets any module that the given module depends on. In general, you need to control related module setting only when debugger performance or memory shortage is a problem.

Related module setting takes place as follows. If M1 is the module that is being set dynamically, then the following modules are considered related and are also set:

- If M1 is a *library body*, the debugger also sets the associated library specification, if any.
- If M1 is a *subunit*, the debugger also sets its parent unit and, therefore, any parent of the parent.
- If M1 mentions a *library package* P1 in a **with** clause, the debugger also sets P1's specification. Neither the body of P1 nor any possible subunits of P1 are set, because symbols declared within them should not be visible outside.

  If P1's specification mentions a package P2 in a **with** clause, the debugger also sets P2's specification. Likewise, if P2's specification mentions a package P3 in a **with** clause, the debugger also sets P3's specification, and so on. The specifications of all such library packages are set so that you can access data components (for example, record components) that may have been declared in other packages.
- If M1 mentions a *library subprogram* in a **with** clause, the debugger does *not* set the subprogram. Only the subprogram name needs to be visible in M1 (no declaration within a library subprogram should be visible outside the subprogram). Therefore, the debugger inserts the name of the library subprogram into the RST when M1 is set.

Thus, after you start a debugging session and type GO to reach the start of the main program, the following modules are set: the main program and any modules related to the main program. As you execute the program under debugger control, modules are set whenever the debugger interrupts the program, such as when a breakpoint occurs.

Once set, a module remains set until its symbol records are removed from the RST by a CANCEL MODULE command (see Section 6.5.2.5).

6.5.2.2 The SHOW MODULE Command

To determine if one or more modules of the program are set, enter the debugger SHOW MODULE command, specifying the module names as parameters. The debugger uses Ada unit-name conventions to distinguish modules internally and in displays. However, the debugger also appends an underscore to specification names to distinguish them from body names (which are the same, according to Ada unit-name conventions). In addition, the debugger recognizes specifications and bodies for library generic instantiations. You should use this notation when specifying module names with the SHOW MODULE command.

The quickest way to determine module names is to use the SHOW MODULE command with no parameters; this command displays information about all of the modules of the program. For example:

```
DBG> SHOW MODULE
module name                          symbols    size

ACCOUNTING                           no         2404
ACCOUNTING_                          no         1492
ADA$ELAB_HOTEL                       yes         236
CONFIRM                              no          768
GUEST_QUEUE                          no         2492
GUEST_QUEUE_                         no         1516
HOTEL                                yes        1068
    .
    .
    .
RESERVATIONS                         no         1252
RESERVATIONS.CANCEL                  no         1100
RESERVATIONS.RESERVE                 no         1104
RESERVATIONS.RESERVE.BILL            no         1064
RESERVATIONS_                        yes         932
    .
    .
    .
total ADA modules: 23.               bytes allocated: 183152.
DBG>
```

The third module displayed (ADA$ELAB_HOTEL) is an internal module used by VAX Ada to elaborate library packages. Its name is formed by prefixing ADA$ELAB_ to the name of the main program (which in this case is HOTEL). You do not need to concern yourself with that module. The remaining modules listed comprise all of the modules for the program. Note the use of the appended underscores to distinguish specifications

(RESERVATIONS_) from bodies (RESERVATIONS) and to identify library generic instantiations (ACCOUNTING_). For each module, the symbols column states whether the module has been set (yes or no); the size column states the size, in bytes, of the module.

After a module has been set, you can reference any symbol defined within that module. See Section 6.4 for information on referring to variables.

6.5.2.3 The SHOW MODULE/RELATED Command

The debugger SHOW MODULE/RELATED command identifies the modules that are related to a specified module as defined in Section 6.5.2.1. The command shows the modules that are automatically set when a given module is set dynamically or with the SET MODULE command (see Section 6.5.2.4). The SHOW MODULE/RELATED command also shows the modules that may be affected when you enter the CANCEL MODULE command (see Section 6.5.2.5).

Consider the following package structure:

```
package P3 is
    . . .
end P3;
----------------------
with P3;
package P1 is
    . . .
end P1;
----------------------
package P2 is
    . . .
end P2;
----------------------
with P2;
package body P1 is
begin
    . . .
end P1;
```

- Package body P1 mentions package specification P2_ in a **with** clause.
- Package specification P1_ mentions package specification P3_ in a **with** clause.

P1_ and P2_ are considered directly related to P1. P3_ is considered related to P1 (by way of P1_).

The SHOW MODULE/RELATED command, applied to package body P1, would display information like the following:

```
DBG> SHOW MODULE/RELATED P1
module name                          symbols     size      relationship

P1                                   yes          868
directly related modules:
  P1_                                yes          884      withed
  P2_                                yes          916      withed
related modules:
  P3_                                yes          868      withed

total ADA modules: 1.                bytes allocated: 109512.
DBG>
```

The entries in the relationship column indicate that all modules directly related and related to P1 are library packages. Note that the debugger treats the relationship between a package body and its specification the same as it treats the relationship between a unit and a package it mentions in a **with** clause. The reason for this nondifferentiation is that the action of making the symbols of a package specification visible elsewhere is the same in both cases.

Consider the following subunit structure:

```
package P4 is
   procedure SUB1;
end P4;
----------------------
package body P4 is
   procedure SUB1 is separate;
end P4;
----------------------
separate(P4)
procedure SUB1 is
   procedure SUB2 is separate;
begin
   . . .
end SUB1;
----------------------
separate(P4.SUB1)
procedure SUB2 is
begin
   . . .
end SUB2;
```

- P4 and P4_ are a library package body and its specification, respectively.
- P4.SUB1.SUB2 is a subunit of P4.SUB1, which is a subunit of P4.

The SHOW MODULE/RELATED command, applied to P4.SUB1, would display information like the following:

```
DBG> SHOW MODULE/RELATED P4.SUB1
module name                    symbols    size     relationship

P4.SUB1                        yes         828
directly related modules:
   P4                          yes         776     parent
   P4.SUB1.SUB2                yes         836     subunit
related modules:
   P4_                         yes         728     withed

total ADA modules: 1.          bytes allocated: 191888.
DBG>
```

The distinction between related and directly related for subunits is analogous to that for library packages.

6.5.2.4 The SET MODULE Command

You use the debugger SET MODULE command if you need to set one or more modules that the debugger has not already set automatically. When using the SET MODULE command, use the naming conventions described in Chapter 1 to specify module (compilation unit) names. You can determine the correct names for modules by entering a SHOW MODULE command.

The following example shows how you might use the SET MODULE command. Procedure COUNTER is local to library package body RESERVATIONS.

```
 . . .
4    package body RESERVATIONS is
 . . .
8       procedure COUNTER (X: in out INTEGER) is
9       begin
10         X := X + 1;
11      end COUNTER;
 . . .
18   end RESERVATIONS;
```

If you try to set a breakpoint on COUNTER before its containing module (RESERVATIONS) has been set, the debugger issues the following message:

```
DBG> SET BREAK COUNTER
%DEBUG-E-NOSYMBOL, symbol 'COUNTER' is not in the symbol table
DBG>
```

Line numbers are symbol information; therefore, you cannot specify line numbers in RESERVATIONS either. For example:

```
DBG> SET BREAK RESERVATIONS.%LINE 7
%DEBUG-I-LINEINFO, no line 7, previous line is 6
%DEBUG-E-NOSYMBOL, symbol '%LINE 7' is not in the symbol table
DBG>
```

Once you have set module RESERVATIONS, you can reference its symbols. For example:

```
DBG> SET MODULE RESERVATIONS
DBG> SET BREAK COUNTER
DBG> SHOW BREAK
breakpoint on ADA Event "DEPENDENTS_EXCEPTION" for any value
breakpoint on ADA Event "EXCEPTION_TERMINATED" for any value
breakpoint at routine RESERVATIONS.COUNTER
DBG>
```

By default, the SET MODULE command sets related modules simultaneously with the module specified in the command. If a related module is already set, then there is no other effect. In the preceding example, setting RESERVATIONS would also set the associated library specification, RESERVATIONS_, because the symbols declared in RESERVATIONS_ must be visible in RESERVATIONS.

Note that you can use the /CALLS qualifier with the SET MODULE command to set all of the modules in the call stack.

You can override the setting of related modules with the /NORELATED command qualifier: SET MODULE/NORELATED sets only the modules you specify explicitly. However, if you use SET MODULE/NORELATED, you may find that a symbol which is declared in another unit and which should be visible at the point of execution is no longer visible; or that a symbol which should be hidden by a redeclaration of that same symbol is now visible.

See Sections 6.5.2.1 and 6.5.2.3 for more information on related modules.

6.5.2.5 The CANCEL MODULE Command

The debugger CANCEL MODULE command deletes the symbol records of one or more specified modules from the RST. When using the CANCEL MODULE command, use the naming conventions described in Chapter 1 to specify module (compilation unit) names. You can determine the correct names for modules by entering a SHOW MODULE command.

CANCEL MODULE/ALL deletes all modules from the RST.

The behavior of the CANCEL MODULE command depends on whether you use the /[NO]RELATED qualifier. CANCEL MODULE/NORELATED deletes from the RST only the modules you specify explicitly.

The effect of CANCEL MODULE/RELATED, which is the default, is to delete related modules in a manner consistent with the intent of Ada's scope and visibility rules. The exact effect depends on module relationships.

For example, consider the following Ada code:

```
package P3 is
    . . .
end P3;
----------------------
with P3;
package P1 is
    . . .
end P1;
----------------------
package body P1 is
begin
    . . .
end P1;
```

P1_ is a library package specification, and P1 is its body. The specification P1_ mentions the library package specification P3_ in a **with** clause. Then:

- CANCEL MODULE/RELATED P3_ deletes only P3_.
- CANCEL MODULE/RELATED P1_ deletes P1_ and P3_ (but P3_ is not deleted if it is directly related to another set module).
- CANCEL MODULE/RELATED P1 deletes P1, P1_, and P3_ (but neither P1_ nor P3_ are deleted if they are directly related to another set module).

Similarly, consider the following set of subunits:

```
package P4 is
   procedure SUB1;
end P4;
----------------------
package body P4 is
   procedure SUB1 is separate;
end P4;
----------------------
separate(P4)
procedure SUB1 is
   procedure SUB2 is separate;
begin
    . . .
end SUB1;
----------------------
separate(P4.SUB1)
procedure SUB2 is
begin
    . . .
end SUB2;
```

P4.SUB1.SUB2 is a subunit of P4.SUB1, which is a subunit of P4. Then:

- CANCEL MODULE/RELATED P4.SUB1.SUB2 deletes P4.SUB1.SUB2.
- CANCEL MODULE/RELATED P4.SUB1 deletes P4.SUB1 and P4.SUB1.SUB2.
- CANCEL MODULE/RELATED P4 deletes P4, P4.SUB1, and P4.SUB1.SUB2.

6.5.3 Resolving Multiply-Defined Symbols

When you reference a multiply-defined symbol in a debugger command, the debugger may not be able to determine the particular declaration of the symbol that you intended. For example:

```
DBG> EXAMINE X
%DEBUG-E-NOUNIQUE, symbol 'X' is not unique
DBG>
```

Alternatively, the debugger may reference a declaration that is visible in the current scope, but the declaration is not the one you are interested in.

To resolve either of these problems, you first need to understand scope, path names, and symbol lookup conventions. Then, you can use the debugger SHOW SYMBOL, SET SCOPE, and SHOW SCOPE commands to help resolve your problems. Scope regions, path names, and symbol lookup are discussed in the following sections. Use of the SHOW SYMBOL, SET

SCOPE, and SHOW SCOPE commands is discussed in Sections 6.5.3.4 and 6.5.3.5.

6.5.3.1 Scope

The *scope* of the subprogram that is executing is called the *PC scope*. When you begin a debugging session, the PC scope is that of the main program. The PC scope is known as scope 0.

As the program executes, the PC scope is redefined so that it is always that of the subprogram that is on top of the call stack. Thus, the scope of the subprogram that called the currently executing subprogram is known as scope 1. Similarly, the scope of its caller is known as scope 2, and so on.

The automatic redefinition of the PC scope follows Ada rules. In some cases, you may need to change the scope; see Section 6.5.3.5 for more information.

6.5.3.2 Path Name Conventions

A *path name* identifies a unique declaration of a symbol. The debugger always uses path names to display symbols; you need to use path names in debugger commands only to resolve ambiguities.

When the debugger language is set to Ada, the debugger generally constructs path names that follow the Ada rules, using selected component notation to separate path name elements (with other languages, a backslash is used to separate elements). However, the debugger also follows the naming conventions described in Chapter 1 to identify specifications, bodies, and subunits. Thus, the following path name identifies the symbol A1, which is declared in the package specification for the unit SYSTEM_OPS:

```
SYSTEM_OPS_.A1
```

The next path name identifies the symbol B1, which is declared in the package body for the unit SYSTEM_OPS:

```
SYSTEM_OPS.B1
```

Because it follows the Ada rules, the debugger also honors **use** clauses. When a **use** clause makes a symbol declared in a package directly visible outside the package, you do not need to specify an expanded name (package-name.symbol) to refer to the symbol in the program itself.

The complete path name format is as follows:

- The leftmost element of a path name identifies the debugger module (Ada compilation unit) containing the symbol. For example, the path

name MAIN.AVERAGE indicates that the symbol AVERAGE is in the compilation unit MAIN.

- Moving toward the right, the path name lists the successively nested subprograms, packages, loop statements, or declare blocks that lead to the particular declaration of the symbol. For example, the path name RESERVATIONS.COUNTER.X indicates that X is in the subprogram COUNTER, which in turn is nested in the package body of the unit RESERVATIONS.
- The rightmost element is the symbol.
- If a label has been assigned to a loop statement or declare block in the source code, the debugger displays the label; otherwise, the debugger displays LOOP$n for a **loop** statement or BLOCK$n for a **declare** block, where n is the line number at which the statement or block begins.

You use path names to refer to symbols that are not visible in the current scope. For example, consider the following source code:

```
package SIMPLE_PKG is
   subtype SMALL_STRING is STRING(1..14);
   S1: SMALL_STRING := "The value is: ";
   I1: INTEGER := 5;
   procedure PRINT_OUT (X: SMALL_STRING; Y: INTEGER);
end SIMPLE_PKG;

with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
package body SIMPLE_PKG is
   procedure PRINT_OUT (X: SMALL_STRING; Y: INTEGER) is
   begin
      PUT(X);
      PUT(Y);
   end PRINT_OUT;
end SIMPLE_PKG;
------------------------------------------------------------------------
with SIMPLE_PKG;
procedure SIMPLE_PROCEDURE is
   S1: SIMPLE_PKG.SMALL_STRING := "What number?? ";
   I1: INTEGER := 6;
begin
   SIMPLE_PKG.PRINT_OUT(S1,I1);
end SIMPLE_PROCEDURE;
```

If the current scope is the procedure SIMPLE_PROCEDURE, then you need to use a path name to examine the variable S1 in the specification of the package SIMPLE_PKG. For example:

```
DBG> EXAMINE SIMPLE_PKG_.S1
SIMPLE_PKG_.S1(1..14):  "The value is: "
DBG>
```

You can also use path names to distinguish among recursive Ada subprograms. In this case, you can include an invocation number in a path name to indicate which call to the subprogram corresponds to the reference. An *invocation number* is a nonnegative integer that you insert in the path name; the number must follow the name of the rightmost subprogram in the path name. Thus, 0 denotes the most recent invocation, 1 denotes the immediately previous invocation, and so on. When you do not include an invocation number, the debugger assumes that the reference is to the most recent call to the subprogram, and it supplies the default value 0.

For example, if FACTORIAL is a recursive function, and each call of FACTORIAL creates a new variable X, then the following path name would indicate X in the most recent call to FACTORIAL:

```
FACTORIAL\FACTORIAL 0\X
```

To refer to the variable X that was generated in the previous call to FACTORIAL, you would express the path name with a 1 in place of the 0. For example:

```
DBG> EXAMINE FACTORIAL\FACTORIAL 1\X
FACTORIAL\FACTORIAL 1.X:       6
DBG>
```

Note that in this case, the debugger recognizes only its backslash notation, rather than the Ada selected component notation.

6.5.3.3 Symbol Lookup Conventions

When you specify a path name for a symbol in a debugger command, the debugger looks for the symbol in the scope region denoted by the path name.

When you do not specify a path name (including an Ada expanded name), the debugger searches the RST as follows. The search proceeds in this order until the specified symbol is found.

1. The debugger looks for the symbol within the module (compilation unit) surrounding the current PC value (where execution is currently suspended).
2. If the symbol is not found, the debugger then searches any package that is mentioned in a **use** clause. The debugger does not distinguish between a library package and a package whose declaration is in the same module as the current scope region. If the same symbol is declared in two or more packages that are visible, the symbol is not unique (according to Ada rules), and the debugger issues a message like the following:

   ```
   %DEBUG-E-NOUNIQUE, symbol 'X' is not unique
   ```

3. If the symbol is still not found, the debugger searches the call stack. It searches scopes 1, 2, 3, ..., N, in that order (as applicable), where N is the number of calls on the call stack. Within each scope region, the search proceeds exactly as described for scope 0 in step 1.
4. Finally, the debugger searches everywhere else in the RST, in all of the modules that have been set. At this point, the debugger does not attempt to resolve multiply-defined symbols. Instead, if more than one occurrence of the symbol is found, the debugger issues a message like the following:

```
%DEBUG-E-NOUNIQUE, symbol 'X' is not unique
```

6.5.3.4 Using the SHOW SYMBOL Command and Path Names to Specify Symbols Uniquely

If the debugger tells you that a symbol reference is not unique, you should use the debugger SHOW SYMBOL command to obtain all possible path names for that symbol. The SHOW SYMBOL command lists all of the possible declarations or definitions of a specified symbol that exist in the RST—that is, in all modules (compilation units) that have been set. For example:

```
DBG> EXAMINE X
%DEBUG-W-NOUNIQUE, symbol 'X' is not unique
DBG> SHOW SYMBOL X
data RESERVATIONS.X
data RESERVATIONS.COUNTER.X
routine QUEUE_MANAGER.X
data SCREEN_IO.INPUT.BUFFER.X
data HOTEL.X
DBG>
```

Note that you can use an asterisk (*) as a wildcard character with the SHOW SYMBOL command—for example, SHOW SYMBOL PL* selects all symbols in the RST that start with the letters PL.

Each path name in the SHOW SYMBOL display identifies a unique declaration of a symbol. In the preceding example, the first two declarations of X are variables (data); the declaration of X in QUEUE_MANAGER is a subprogram or function (routine). Each declaration of X includes its path name prefix, which indicates the path (search scope) the debugger must follow to reach that particular declaration. (See Sections 6.5.3.1 through 6.5.3.3 for more information on scope regions, path names, and symbol lookup.)

After you have identified the path name you want to reference, you just use it in your command. For example:

```
DBG> EXAMINE RESERVATIONS.X
RESERVATIONS.X: 3
DBG>
```

Path names may include line numbers. For example:

```
DBG> SET BREAK RESERVATIONS.%LINE 14 DO (EXAMINE RESERVATIONS.X)
DBG> GO
break at RESERVATIONS.%LINE 14
    14:    CANCEL(X);
RESERVATIONS.X: 3
```

When displaying symbols, the debugger always uses path names; but you need to use path names in debugger commands only to resolve an ambiguity.

When it displays path names, the debugger uses the file-name conventions described in Chapter 1 to identify specifications, bodies, and subunits. Thus, in the preceding example, HOTEL, RESERVATIONS, SCREEN_IO, and QUEUE_MANAGER are bodies, and BUFFER is a subunit of INPUT, which is itself a subunit of SCREEN_IO.

In general, you should specify path names exactly as indicated in the SHOW SYMBOL display. But you can also take shortcuts. For example, unless there is a possible ambiguity, you do not have to type the trailing underscore character to distinguish a specification from its body. The debugger can usually distinguish the two from the context. You can also shorten, or abbreviate, path names. When abbreviating, start from the left, leaving enough of the path name to uniquely specify it. For example, BUFFER.X is a valid abbreviated path name for SCREEN_IO.INPUT.BUFFER.X.

A path name can be regarded as the scope for one symbol, or for a set of symbols. For example, the following command identifies all of the declarations in QUEUE_MANAGER:

```
DBG> SHOW SYMBOL * IN QUEUE_MANAGER
package body QUEUE_MANAGER
routine QUEUE_MANAGER.PRINT
routine QUEUE_MANAGER.X
DBG>
```

If you want to make frequent references to a symbol with a long path name, you can define a new symbol for it with the DEFINE command. For example:

```
DBG> DEFINE Q_X = QUEUE_MANAGER.PRINT
DBG>
```

In Ada, a **use** clause makes a symbol declared in a package directly visible outside the package. In such cases, you do not need to specify an expanded name (package-name.symbol) to refer to the symbol in the program itself. This is also true when you reference the symbol in a debugger command. The SHOW SYMBOL/USE_CLAUSE command identifies any package (library or otherwise) that a specified block, subprogram, or package mentions in a **use** clause. If the entity specified is a package (library or otherwise), the command also identifies any block, subprogram, package, and so on, that the specified module is mentioned by in a **use** clause. For example:

```
package A is
   X: INTEGER;
 . . .
end A;

with A;use A;
package B is
   Y: INTEGER;
 . . .
end B;

with B;
function F return BOOLEAN is
use B;
 . . .
end F;

DBG>  SHOW SYMBOL/USE_CLAUSE B_
package spec B_
    used by:  F
    uses:     A_
DBG>
```

Thus, when the current scope is B, you can reference X in A without having to use an expanded name, just as in the program.

Refer to Section 6.5.3.2 and the *VMS Debugger Manual* for more information on the use of path names.

6.5.3.5 Using the SET SCOPE Command to Specify a Symbol Search Scope

The debugger SET SCOPE command enables you to establish a new scope for symbol lookup, so that you do not have to use a path name when referencing symbols in that scope region. After you have entered a SET SCOPE command, the debugger applies the path name you specified in the command to all references that are not individually qualified with path names.

The following command sets the current scope to that of routine QUEUE_MANAGER.PRINT:

```
DBG> SET SCOPE QUEUE_MANAGER.PRINT
DBG>
```

You can use numeric path names with the SET SCOPE command. Numeric path names refer to the order of calls on the call stack (as displayed by SHOW CALLS). The number 0 means the PC scope; 1 means the scope of the caller subprogram; 2 means the scope of the caller's caller, and so on. For example, the following command sets the current scope to be three calls down from the PC scope:

```
DBG> SET SCOPE 3
DBG>
```

You can also define a scope search list to specify the order in which the debugger should search for symbols. The debugger's default scope search list is equivalent to entering the following hypothetical command:

```
DBG> SET SCOPE 0,1,2,3,...N
```

Here the debugger searches successively down the call stack (0 is the PC scope, 1 is the scope of the caller subprogram, and so on).

The following command instructs the debugger to search first for symbols in the PC scope (denoted by 0). If the debugger cannot find a specified symbol using that scope, it uses QUEUE_MANAGER.PRINT; if necessary, it then uses SCREEN_IO.INPUT.BUFFER; and if further necessary, it uses the entire RST until it finds a definition.

```
DBG> SET SCOPE 0, QUEUE_MANAGER.PRINT, SCREEN_IO.INPUT.BUFFER
DBG>
```

The scope defined in a SET SCOPE command becomes the default scope (or scope search list) for all symbol searches until you explicitly change or cancel the scope. You can determine the current scope at any time by entering the SHOW SCOPE command. For example:

```
DBG> SHOW SCOPE
scope:
    0 [ = HOTEL],
    QUEUE_MANAGER.PRINT,
    SCREEN_IO.INPUT.BUFFER
DBG>
```

If no SET SCOPE command has been entered, the SHOW SCOPE command responds as in the following example:

```
DBG> SHOW SCOPE
scope:
    0 [ = HOTEL ],
    1 [ = ADA$ELAB_HOTEL ],
    2,
    3,
    4 [ = ADA$ELAB_HOTEL\ADA$ELAB_HOTEL 1 ],
    5
DBG>
```

The number displayed in the SHOW SCOPE command (0, 1, 2, and so on) indicates the call stack: 0 is the PC scope. The brackets enclose the names of the modules and (if applicable) subprograms.

The CANCEL SCOPE command resets the default scope search list (0, 1, 2,...N).

6.5.4 Resolving Overloaded Names and Symbols

When you encounter overloaded names and symbols, the debugger issues a message like the following:

```
%DEBUG-E-NOTUNQOVR, symbol 'ADD' is overloaded
        use SHOW SYMBOL to find the unique symbol names
```

If the overloaded symbol is an enumeration literal, you can use qualified expressions to resolve the overloadings. See Section 6.4.7.1 for an example of using qualified expressions.

If the overloaded symbol represents a subprogram or task accept statement, you can use the unique name generated by the compiler for the debugger. The compiler always generates unique names for subprograms declared in library package specifications, because the names might later be overloaded in the package body. Unique names are only generated for task accept statements and subprograms declared in other places if the task accept statements or subprograms are actually overloaded.

Overloaded task accept statement names and subprogram names are distinguished by a suffix consisting of two underscores followed by an integer that uniquely identifies the given symbol. You must use the unique naming notation in debugger commands to uniquely specify a subprogram whose name is overloaded. However, if there is no ambiguity, you do not need to use the unique name, even though one was generated.

For example, suppose you are debugging a library package with the following declarations:

```
package SYSTEM_OPS is
   type A1 is array (1..10) of INTEGER;
   type A2 is array (1..20) of INTEGER;
   function ADD(X,Y: A1) return A1;
   function ADD(X,Y: A2) return A2;

   function DIVIDE (X,Y: INTEGER) return INTEGER;
end SYSTEM_OPS;

package body SYSTEM_OPS is
   function ADD(X,Y: A1) return A1 is
   begin
      return (1..10 => 3);
   end;

   function ADD(X,Y: A2) return A2 is
   begin
      return (1..20 => 5);
   end;

   function DIVIDE (X,Y: INTEGER) return INTEGER is
   begin
      return X/Y;
   end;

   function SQUARE (X: INTEGER) return INTEGER is
   begin
      return X*X;
   end;

   function SQUARE (X: FLOAT) return FLOAT is
   begin
      return X*X;
   end;
 . . .
end SYSTEM_OPS;
```

If you try to set a breakpoint on ADD (declared in the package specification), you will receive an error like the following:

```
%DEBUG-E-NOTUNQOVR, symbol 'ADD' is overloaded
        use SHOW SYMBOL to find the unique symbol names
```

Then, when you enter the debugger SHOW SYMBOL command, you receive a list of the overloaded symbols and their origins (SYSTEM_OPS_ is the package specification; SYSTEM_OPS is the package body):

```
DBG> SHOW SYMBOL ADD
overloaded symbol SYSTEM_OPS.ADD
  overloaded instance SYSTEM_OPS.ADD__1
  overloaded instance SYSTEM_OPS.ADD__2
DBG>
```

Similarly, if you try to set a breakpoint on SQUARE (declared in the body of the package SYSTEM_OPS), you receive a similar set of responses:

```
DBG> SET BREAK SQUARE
%DEBUG-E-NOTUNQOVR, symbol 'SQUARE' is overloaded
        use SHOW SYMBOL to find the unique symbol names
DBG> SHOW SYMBOL SQUARE
overloaded symbol SYSTEM_OPS.SQUARE
  overloaded instance SYSTEM_OPS.SQUARE__1
  overloaded instance SYSTEM_OPS.SQUARE__2
DBG>
```

You can use the debugger EXAMINE/SOURCE command to determine which declaration an overloaded subprogram suffix number corresponds to. For example:

```
DBG> EXAMINE/SOURCE SYSTEM_OPS.ADD__1, SQUARE__1, SQUARE__2
module SYSTEM_OPS
     5:     function ADD(X,Y: A1) return A1 is
module SYSTEM_OPS
    20:     function SQUARE (X: INTEGER) return INTEGER is
module SYSTEM_OPS
    25:     function SQUARE (X: FLOAT) return FLOAT is
DBG>
```

You can then uniquely specify a particular declaration of an overloaded name. For example:

```
DBG> SET BREAK SYSTEM_OPS.ADD__1, SQUARE__2
DBG>
```

6.6 Supplementary Debugger Features

This section presents additional debugger features that you may find useful. The following subjects are covered briefly:

- Logging a debugging session into a file
- Invoking an editor from the debugger
- Using a debugger initialization file
- Using command procedures to control debugging sessions
- Using the CALL command

See the *VMS Debugger Manual* for more information on all supplementary debugger features.

6.6.1 Logging a Debugging Session into a File

A debugger log file maintains a history of a debugging session. Each debugger command entered during the session and the subsequent debugger output are stored in the file.

Note that the DBG> prompt is not recorded in the debugger log file, and the debugger output is commented out with exclamation points so that the file can be used as a debugger command procedure. Thus, if a lengthy debugging session is interrupted, you can execute the log file as you would any other debugger command procedure, and it will restore the debugging session to the point at which it was previously terminated.

To create a log file, specify the following debugger command:

```
DBG> SET OUTPUT LOG
```

You may want to enter the SET OUTPUT LOG command in your debugger initialization file (see Section 6.6.3). The SHOW OUTPUT command indicates whether you are logging the session.

By default, the output is written to the file DEBUG.LOG in your current default directory, but you can provide a different file specification for the log file. For example, the following command lines specify the file PROCESSOR_V1.LOG as the debug log file:

```
DBG> SET OUTPUT LOG
DBG> SET LOG PROCESSOR_V1.LOG
DBG> SHOW OUTPUT
noverify, terminal, noscreen_log, logging to
DISK:[JONES.WORK]PROCESSOR_V1.LOG;2
```

Note that the debugger logs output only after you have entered a SET OUTPUT LOG command. You can use the SET LOG command to change the name of the log file at any time. In this example, the SET OUTPUT LOG command logs the SET LOG command to the file DEBUG.LOG. Then it closes DEBUG.LOG and starts logging output in the file PROCESSOR_V1.LOG.

6.6.2 Invoking an Editor from the Debugger

The debugger EDIT command allows you to invoke an editor from the debugger. By default, the VAX Language-Sensitive Editor is invoked if it is installed on your system. You can use the debugger SET EDIT command to establish another editor (see Chapter 1 for summary information about the available editors).

The EDIT command allows you to correct errors in your source file as you discover them during a debugging session, without losing the debugger execution context. When you exit from the editor, you return to the debugger prompt, at the same program location where you entered the EDIT command.

By default, when you use the EDIT command, the debugger fetches the external source file that was compiled to produce the currently executing compilation unit. You do not edit the copied source file that the debugger displays in screen mode (see Section 6.2.4.2).

The file specifications of the source files you edit are embedded in the associated object files during compilation (unless you specify /NODEBUG). If some source files have been relocated since the units being debugged were compiled, the debugger may not find them.

In such cases, you can use the debugger SET SOURCE/EDIT command to specify a search list of one or more directories where the debugger should look for source files. For example, the following command line instructs the debugger to look for source files first in the current default directory, and then in directory USER:[JONES.HOTEL]:

```
DBG> SET SOURCE/EDIT [],USER:[JONES.HOTEL]
```

You can also provide file specifications rather than directory specifications with the SET SOURCE/EDIT command. The SET SOURCE/EDIT command does not affect the search list for copied source files that the debugger displays in screen mode.

The SHOW SOURCE/EDIT command displays the source-file search list currently being used for the EDIT command. The CANCEL SOURCE/EDIT command cancels the source-file search list currently being used for the EDIT command and restores the default search mode.

6.6.3 Using a Debugger Initialization File

You can use a command procedure as a debugger initialization file by equating it to the process logical name DBG$INIT. The file assigned the name DBG$INIT automatically executes when you invoke the debugger.

The following command procedure contains sample commands used to initialize a debugging session:

```
USER:[JONES.HOTEL]DEBUGSTART.COM

! If source is not in current default directory, use [SMITH.SHARE]
SET SOURCE [],[SMITH.SHARE]
SET OUTPUT LOG,VERIFY
SET MODE SCREEN
```

To make DEBUGSTART.COM the debugger initialization file, you equate it to DBG$INIT using either the DCL ASSIGN or DEFINE command. For example:

```
$ ASSIGN USER:[JONES.HOTEL]DEBUGSTART.COM DBG$INIT
$ DEFINE DBG$INIT USER:[JONES.HOTEL]DEBUGSTART.COM
```

A sample debugger initialization file for VAX Ada tasking programs is shown in Chapter 7.

6.6.4 Using Command Procedures to Control Debugging Sessions

Like the VMS command-line interpreter, the debugger can execute a sequence of debugger commands contained in a file—a debugger command procedure. The execution syntax is the same as for any series of DCL commands. For example, the following command line invokes the command procedure TEST.COM (.COM is the default file type for command procedures):

```
DBG> @TEST
```

You can execute a debugger command procedure interactively from within a DO command sequence, or from within another command procedure. Command procedures are especially useful when you regularly perform a number of standard setup debugger commands, as specified in a debugger initialization file (see Section 6.6.3). You can also use a debugger log file as a command procedure. For example:

```
DBG> SET OUTPUT VERIFY
DBG> @PROCESSOR_V1.LOG
%DEBUG-I-VERIFYICF, entering command procedure PROCESSOR_V1.LOG
    .
    .
    .
%DEBUG-I-VERIFYICF, exiting command procedure PROCESSOR_V1.LOG
DBG>
```

As shown in this example, when using a log file as a command procedure, you can first enter the SET OUTPUT VERIFY command so that debugger commands are displayed as they are entered and executed.

See Section 6.6.1 for information on how to generate a debugger log file.

6.6.5 The CALL Command

The debugger CALL command performs the following steps:

1. Invokes a subprogram, passing it any specified parameters
2. Executes the subprogram
3. Displays the value returned for a function in register R0 (the value returned for a procedure is 0)

The general form of the CALL command is as follows:

```
DBG> CALL subprogram-name[(parameter[,parameter,...])]
```

When debugging VAX Ada programs, you can use the CALL command reliably only with a subprogram that has been exported. An exported subprogram must be a library subprogram or must be declared in the outermost declarative part of a library package.

The CALL command does not check whether or not the subprogram can be exported, nor does it check the parameter-passing mechanisms that you specify. Note that you cannot use the CALL command to modify the value of a parameter.

A CALL command may result in a deadlock if it is entered when the VAX Ada run-time library is executing. The VAX Ada run-time library routines acquire and release internal locks that allow the routines to operate in a tasking environment. Deadlock can result if a subprogram called from the CALL command requires a resource that has been locked by an executing VAX Ada run-time library routine. To avoid this situation in a nontasking program, enter the CALL command immediately before or after an Ada statement has been executed. However, this approach is not sufficient to assure that deadlock will not occur in a tasking program, as some other task may be executing a VAX Ada run-time library routine at the time of the call. If you must use the CALL command in a tasking program, you can avoid deadlock if the called subprogram does not do any tasking or input-output operations.

See the *VMS Debugger Manual* for additional details on using the CALL command. See Chapter 7 for more information on task debugging.

6.7 Sample Debugging Session

This section shows a sample debugging session with a VAX Ada program, ADD_INTEGERS, that contains a logic error. Line numbers have been added to facilitate the discussion.

```
 1   with TEXT_IO; use TEXT_IO;
 2   with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
 3   procedure ADD_INTEGERS is
 4      HIGHEST,TOTAL: INTEGER;
 5  begin
 6      TOTAL := 0;
 7      loop
 8         PUT("Type a number greater than 0, or 0 to quit: ");
 9         GET(HIGHEST);
10         if HIGHEST <= 0 then
11            exit;
12         else
13            for I in 1..HIGHEST loop
14               TOTAL := TOTAL + I;
15           end loop;
16         end if;
17         PUT("The sum of integers from 1 through ");
18         PUT(HIGHEST);
19         PUT(" is ");
20         PUT(TOTAL);
21         NEW_LINE;
22     end loop;
23   end ADD_INTEGERS;
```

This program prompts for a number and prints the sum of the integers from 1 through the number entered. The problem in the program occurs because the variable TOTAL is not reinitialized when a new number is entered; the statement assigning the value 0 to TOTAL occurs before the loop instead of within it.

Initially, you might compile, link, and run the program as follows:

```
$ ADA ADD_INTEGERS
$ ACS LINK ADD_INTEGERS
$ RUN ADD_INTEGERS
Type a number greater than 0, or 0 to quit: 5
The sum of integers from 1 through           5 is          15
Type a number greater than 0, or 0 to quit: 4
The sum of integers from 1 through           4 is          25
Type a number greater than 0, or 0 to quit: 0
$
```

The program returns a correct sum for the first number you enter, but the sum for the second number is obviously too high.

To debug the program, you must compile and link with the debugger. If you want a listing with line numbers to refer to during the debugging session, include the /LIST qualifier with the ADA command, and then print the listing file that results. For example:

```
$ ADA/DEBUG/LIST/NOOPTIMIZE ADD_INTEGERS
$ ACS LINK/DEBUG ADD_INTEGERS
$ PRINT ADD_INTEGERS.LIS
```

You are now ready to begin a debugging session. The terminal session is keyed to the numbered notes that follow.

```
$ RUN ADD_INTEGERS

              VAX DEBUG Version V5.0-00

%DEBUG-I-INITIAL, language is ADA, module set to ADD_INTEGERS
%DEBUG-I-NOTATMAIN, type GO to get to start of main program  (1)
DBG> SET BREAK %LINE 7    (2)
DBG> GO    (3)
break at routine ADD_INTEGERS
     3: procedure ADD_INTEGERS is
DBG> GO
break at ADD_INTEGERS.LOOP$7.%LINE 8    (4)
     8:       PUT("Type a number greater than 0, or 0 to quit: ");
DBG> EXAMINE TOTAL
ADD_INTEGERS.TOTAL:      0    (5)
DBG> GO
Type a number greater than 0, or 0 to quit: 5
The sum of integers from 1 through           5 is          15    (6)
break at ADD_INTEGERS.LOOP$7.%LINE 8
     8:       PUT("Type a number greater than 0, or 0 to quit: ");    (7)
DBG> EXAMINE TOTAL
ADD_INTEGERS.TOTAL:      15    (8)
DBG> DEPOSIT TOTAL := 0    (9)
DBG> GO
Type a number greater than 0, or 0 to quit: 4
The sum of integers from 1 through           4 is          10    (10)
break at ADD_INTEGERS.LOOP$7.%LINE 8
     8:       PUT("Type a number greater than 0, or 0 to quit: ");
DBG> GO
Type a number greater than 0, or 0 to quit: 0  (11)
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EXIT    (12)
$
```

The actions in this example are keyed to the following notes:

(1) When you enter the RUN command, the debugger displays an informational message and the DBG> prompt. You are now in the default

noscreen mode. The lines of source code are displayed as they are executed, by default.

❷ You decide that the problem may lie with the initialization of the variable TOTAL. You can test this hypothesis by examining the value of TOTAL each time you enter a new number. To stop the program at the point at which you can do this, you set a breakpoint at the line that begins the loop (%LINE 7).

❸ The first GO command executes the program's elaboration code, and breaks at the main program; the next GO command starts program execution.

❹ When the loop statement is reached, the debugger interrupts program execution and displays the source line at which the breakpoint was set. Note that the debugger interrupts execution only at executable lines; thus, the break occurs at the first line inside the loop, not at the loop statement.

❺ Use the EXAMINE command to determine the current value of the variable TOTAL. Its value is 0, as expected at this point.

❻ The GO command resumes program execution. The program now prompts you for a number. You type 5. The program's response is correct.

❼ The debugger again reaches the breakpoint at the first executable line inside the loop and displays the source line.

❽ You examine the variable TOTAL with the EXAMINE command. Its value is 15, not 0 as it should be. This indicates that the assignment statement that initializes TOTAL is misplaced.

❾ The DEPOSIT command replaces the contents of TOTAL with 0, which allows the program to return a correct result the next time through the loop.

❿ The GO command resumes program execution. The result is correct.

⓫ When you enter a 0 in response to the prompt, the program exits, causing the debugger to display a message that indicates the termination status.

⓬ The EXIT command terminates the debugging session.

You can now correct the program so that it reinitializes the variable TOTAL correctly.

Chapter 7

Debugging VAX Ada Tasks

All of the debugger techniques covered in Chapter 6 apply to tasks. However, the debugger provides additional features that allow you to observe task characteristics, control task states, and monitor events that are specific to tasks, such as rendezvous. For example:

- The debugger SHOW TASK command allows you to observe task states and the tasks in your program in detail.
- The debugger SET TASK command allows you to control execution rates and task ordering by setting task states, priorities, time-slicing values, and so on.
- The debugger SET BREAK/EVENT and SET TRACE/EVENT commands allow you to monitor a variety of tasking events and state transitions.

This chapter describes how to use these additional features. You should be familiar with the tasking information in the *VAX Ada Language Reference Manual* and *VAX Ada Run-Time Reference Manual.*

When using these features, remember that use of the debugger may alter the behavior of a tasking program from run to run. For example, while you are suspending execution of the currently active task at a breakpoint, the delivery of an AST (asynchronous system trap) as some input-output completes may make some other task eligible to run as soon as you allow execution to continue.

7.1 A Sample Tasking Program

Example 7–1 demonstrates a number of common errors that you may encounter when debugging tasking programs. The labels (<<B1>>, and so on) in the example mark points of interest where breakpoints could be set and the state of each task observed. If you were to run the example under debugger control, you could enter the following command to set breakpoints at each label and display the current state of each task:

```
DBG> SET BREAK B1,B2,B3,B4,B5,B6,B7 DO (SHOW TASK/ALL)
```

The program creates four tasks:

- An environment task that runs the main program, TASK_EXAMPLE. This task is created before any library packages are elaborated (in this case, TEXT_IO). The environment task has the task ID %TASK 1 in the SHOW TASK displays.
- A task object named FATHER. This task is declared by the main program, and is designated %TASK 2 in the SHOW TASK displays.
- A single task named MOTHER. This task is declared by the main program, and is designated %TASK 3 in the SHOW TASK displays.
- A single task named CHILD. This task is declared by task FATHER, and is designated %TASK 4 in the SHOW TASK displays.

Example 7–1: Procedure TASK_EXAMPLE

```
 1 -- Tasking program that demonstrates various tasking conditions.
 2
 3 with TEXT_IO; use TEXT_IO;
 4 procedure TASK_EXAMPLE is  ❶
 5
 6    pragma TIME_SLICE(0.0); -- Disable time slicing. ❷
 7
 8    task type FATHER_TYPE is
 9       entry START;
10       entry RENDEZVOUS;
11       entry BOGUS; -- Never accepted, caller deadlocks.
12    end FATHER_TYPE;
13
14    FATHER : FATHER_TYPE; ❸
15
```

(continued on next page)

Example 7–1 (Cont.): Procedure TASK_EXAMPLE

```
16    task body FATHER_TYPE is
17       SOME_ERROR : exception;
18
19       task CHILD is  ❹
20          entry E;
21       end CHILD;
22
23       task body CHILD is
24       begin
25          FATHER_TYPE.BOGUS;
26       end CHILD;
27
28       --  CHILD deadlocks on call to its parent
29       --  (parent does not have an accept
30       --  statement for entry BOGUS).
31
32    begin -- (of FATHER_TYPE body)
33
34       accept START do
35 <<B1>>    -- Main program is waiting for this rendezvous to
36          -- complete; CHILD is suspended when it calls the
37          -- entry BOGUS.
38           null;
39       end START;
40
41       PUT_LINE("FATHER is now active and");  ❺
42       PUT_LINE("is going to rendezvous with main program.");
43
44       for I in 1..2 loop
45          select
46             accept RENDEZVOUS do
47                PUT_LINE("FATHER now in rendezvous with main program");
48             end RENDEZVOUS;
49          or
50             terminate;
51          end select;
52
53          if I = 2 then
54             raise SOME_ERROR;
55          end if;
56       end loop;
57
```

(continued on next page)

Example 7–1 (Cont.): Procedure TASK_EXAMPLE

```
 58    exception
 59       when others =>
 60 <<B2>>    -- CHILD is suspended on entry call to BOGUS.
 61          -- Main program is going to delay while FATHER terminates.
 62          -- MOTHER is ready to begin executing.
 63          abort CHILD;
 64 <<B3>>    -- CHILD is now abnormal due to the abort statement.
 65
 66          raise; -- SOME_ERROR exception terminates FATHER.
 67    end FATHER_TYPE;
 68
 69    task MOTHER is    ❻
 70       entry START;
 71       pragma PRIORITY (6);
 72    end MOTHER;
 73
 74    task body MOTHER is
 75    begin
 76       accept START;
 77 <<B4>>    -- At this point, the main program is waiting for its
 78          -- dependents (FATHER and MOTHER) to terminate. FATHER
 79          -- is terminated.
 80       null;
 81    end MOTHER;
 82
 83 begin     -- (of TASK_EXAMPLE)    ❼
 84 <<B5>>    -- FATHER is suspended at accept start.
 85          -- CHILD is suspended in its deadlock.
 86          -- MOTHER has activated and is ready to begin executing.
 87    FATHER.START;          ❽
 88 <<B6>>    -- FATHER is suspended at its 'select or terminate'
 89          -- statement.
 90
 91
 92    FATHER.RENDEZVOUS;    ❾
 93    FATHER.RENDEZVOUS;    ❿
 94    loop                  ⓫
 95       -- This loop causes the main program to busy wait
 96       -- for the termination of FATHER, so that FATHER
 97       -- can be observed in its terminated state.
 98       if FATHER'TERMINATED then
 99          exit;
100       end if;
101       delay 1.0;
102    end loop;
```

(continued on next page)

Example 7–1 (Cont.): Procedure TASK_EXAMPLE

```
103 <<B7>>    -- FATHER has terminated by now with the unhandled
104           -- exception SOME_ERROR. CHILD no longer exists
105           -- because its master (FATHER) has terminated. Task
106           -- MOTHER is ready.
107     MOTHER.START;       ⓬
108     -- The main program enters a wait-for-dependents state,
109     -- so that MOTHER can finish executing.
110 end TASK_EXAMPLE;       ⓭
111
```

Key to Example 7–1:

❶ After all of the library packages are elaborated (in this case, TEXT_IO), the main program is automatically called and begins to elaborate its declarative part (lines 5 through 82).

❷ To ensure repeatability from run to run, the example uses no time slicing. The 0.0 value for the pragma TIME_SLICE documents that the procedure TASK_EXAMPLE needs to have time slicing disabled (time slicing is disabled if the pragma TIME_SLICE is omitted or is specified with a value of 0.0).

❸ Task object FATHER is elaborated, and a task designated %TASK 2 is created. FATHER (%TASK 2) is created in a suspended state and is not activated until the beginning of the statement part of the main program (line 83), in accordance with Ada rules. The elaboration of the task body on lines 16 through 67 defines the statements that tasks of type FATHER_TYPE will execute.

❹ Task FATHER declares a single task named CHILD (line 19). A single task represents both a task object and an anonymous task type. Task CHILD is not created or activated until FATHER is activated.

❺ The only source of ASTs is this series of TEXT_IO.PUT_LINE statements (input-output completion delivers ASTs).

❻ A single task, MOTHER, is defined, and a task designated %TASK 3 is created. The pragma PRIORITY gives MOTHER a priority of 6.

❼ The tasks FATHER and MOTHER are activated in parallel, while the main program waits. FATHER has no pragma PRIORITY, and thus assumes a default priority of 7. Because this is higher than the priority of MOTHER, FATHER executes its activation first. Its activation consists of the elaboration of lines 16 to 31.

When task FATHER is activated, it waits while its task CHILD is activated and a task designated %TASK 4 is created. CHILD executes one entry call on line 25, and then deadlocks because the entry is never accepted.

❽ This is the first rendezvous the main program makes with task FATHER. This rendezvous causes FATHER to suspend at its first accept statement (line 34). Note that FATHER continues to execute past the end of its activation, even though MOTHER has not been activated, because VAX Ada attempts to continue tasks as far as they will go, to minimize task switch overhead. When FATHER becomes suspended, MOTHER begins its activation, and executes lines 74 and 75.

❾ After tasks FATHER and MOTHER are activated, the main program (%TASK 1) is eligible to resume its execution. Because %TASK 1 has the default priority of 7, which is higher than MOTHER's priority, the main program resumes execution.

❿ At the third rendezvous with FATHER, FATHER raises the exception SOME_ERROR on line 54. The handler on line 59 catches the exception, aborts the suspended CHILD task, and then reraises the exception; FATHER then terminates.

⓫ A loop with a delay statement ensures that when control reaches line 103, FATHER has executed far enough to be terminated.

⓬ This entry call ensures that MOTHER does not wait forever for its rendezvous on line 76. MOTHER executes the accept statement (which involves no other statements), the rendezvous is completed, and MOTHER is immediately switched off the processor at line 77 because its priority is only 6.

⓭ After its rendezvous with MOTHER, the main program (%TASK 1) executes lines 108 through 110. At line 110, the main program must wait for all its dependent tasks to terminate. When the main program reaches line 110, the only nonterminated task is MOTHER (MOTHER cannot terminate until the **null** statement at line 80 has been executed). MOTHER finally executes to its completion at line 81. Now that all tasks are terminated, the main program completes its execution. The main program then returns and execution resumes with the VMS command-line interpreter.

7.2 Referring to Tasks in Debugger Commands

You refer to tasks in debugger commands using three kinds of expressions:

- An Ada language expression for a task value (for example, FATHER)
- A task ID (for example, %TASK 2)
- A pseudotask name (for example, %ACTIVE_TASK)

You can mix these expressions in the same debugger command line.

The following sections discuss these expressions in more detail and give examples of how to use them (the examples are derived from Example 7–1).

NOTE

The debugger does not support the task-specific attributes T'CALLABLE, E'COUNT, T'STORAGE_SIZE, and T'TERMINATED. See Section 7.2.4 for more information.

7.2.1 Ada Language Expressions for Tasks

A *task* is an entity that executes in parallel with other tasks. A task is characterized by a unique task ID (defined in Section 7.2.2), a separate stack, and a separate register set. You declare a task either by declaring a single task or by declaring an object of a task type. For example:

```
-- TASK TYPE declaration.
--
task type FATHER_TYPE is
 . . .
end FATHER_TYPE;

task body FATHER_TYPE is
 . . .
end FATHER_TYPE;

-- A single task.
--
task MOTHER is
 . . .
end MOTHER;

task body MOTHER is
 . . .
end MOTHER;
```

A *task object* is a data item that contains a 32-bit task value. A task object is created when the program elaborates a single task or task object, when you declare a record or array containing a task component, or when a task allocator is evaluated. For example:

```
-- Task object declaration.
--
FATHER : FATHER_TYPE;

-- Task object (T) as a component of a record.
--
type SOME_RECORD_TYPE is
   record
      A, B: INTEGER;
      T   : FATHER_TYPE;
   end record;

HAS_TASK : SOME_RECORD_TYPE;

-- Task object (POINTER1) via allocator.
--
type A is access FATHER_TYPE;
POINTER1 : A := new FATHER_TYPE;
```

A task object is comparable to any other object. You refer to a task object in debugger commands either by name or by path name. For example:

```
DBG> EXAMINE FATHER
DBG> EXAMINE FATHER_TYPE$TASK_BODY.CHILD
```

See Chapter 6 for more information on path names.

When a task object is elaborated, a task is created by the VAX Ada run-time library, and the task object is assigned its 32-bit task value. As with other Ada objects, the value of a task object is undefined before the object is initialized, and the results of using an uninitialized value are unpredictable.

The *task body* of a task type or single task is implemented in VAX Ada as a procedure. This procedure is called by the VAX Ada run-time library when a task of that type is activated. A task body is treated by the debugger as a normal Ada procedure, except that it has a specially constructed name.

To specify the task body in a debugger command, use the following syntax to refer to tasks declared as task types:

```
task-type-identifier$TASK_BODY
```

Use the following syntax to refer to single tasks:

```
task-identifier$TASK_BODY
```

For example:

```
DBG> SET BREAK FATHER_TYPE$TASK_BODY
```

7.2.2 Task ID (%TASK)

A *task ID* is the value used by the VAX Ada run-time library and debugger to uniquely identify a task during the entire execution of a program.

A task ID has the following syntax, where n is a positive decimal integer:

```
%TASK n
```

You can determine the task ID of a task object by evaluating or examining the task object. For example:

```
DBG> EVALUATE FATHER
%TASK 2
DBG> EXAMINE FATHER
TASK_EXAMPLE.FATHER:  %TASK 2
```

You can also use the SHOW TASK/ALL command to identify the task IDs that have been assigned to all currently existing tasks. For example:

```
DBG> SHOW TASK/ALL

  task id pri hold state   substate    task object
* %TASK 1  7       RUN               SHARE$ADARTL+130428
  %TASK 2  7       SUSP  Accept      TASK_EXAMPLE.MOTHER+4
  %TASK 4  7       SUSP  Entry call TASK_EXAMPLE.FATHER_TYPE$TASK_BODY.CHILD+4
  %TASK 3  6       READY             TASK_EXAMPLE.MOTHER+4

DBG>
```

You can use task IDs to refer to nonexistent tasks in debugger conditional statements. For example, if you had already run your program once, and you discovered that %TASK 2 and 3 were of interest, you could enter the following commands at the beginning of your next debugging session, before %TASK 2 or 3 was created:

```
DBG> SET BREAK %LINE 44 WHEN (%ACTIVE_TASK=%TASK 2)
DBG> IF (%CALLER=%TASK 3) THEN (SHOW TASK/FULL)
```

In other words, you can use a task ID in certain debugger commands before the task has been created, without the debugger reporting an error (as it would if you were to use a task object name before the task object came into existence). A task does not exist until the task object is elaborated, and later becomes nonexistent sometime after it terminates (when the task's master terminates). A nonexistent task never appears in a debugger SHOW TASK display.

Each time a program is run, the same task IDs are assigned to the same tasks as long as the program statements are executed in the same order. Different execution orders may result from asynchronous system traps (ASTs) (caused by **delay** statement expiration or input-output completion) being delivered in a different order or from time slicing being enabled. Task IDs are never reassigned during the execution of the program.

The VAX Ada run-time library always assigns %TASK 1 to the environment task that executes the main program; it always assigns %TASK 0 to the null task that executes when there are no other tasks—including the main program—eligible to execute. The null task is a special task created by the run-time library; you cannot apply most debugger commands to the null task.

7.2.3 Pseudotask Names

The debugger recognizes a number of significant tasks by pseudotask name:

- %ACTIVE_TASK—refers to the task that will run when a STEP or GO command is executed
- %VISIBLE_TASK—refers to the task whose task and register set are the current context for looking up names, calls, and so on
- %NEXT_TASK—refers to the task that will run next, after the active task
- %CALLER_TASK—when an accept statement is being executed, refers to the task that called the entry associated with the accept statement

More information on these pseudotask names and examples of their use with various debugger commands are given in the following sections.

7.2.3.1 Active Task (%ACTIVE_TASK)

The *active task* is the task that runs when a debugger STEP or GO command is executed. Initially, it is the task that is interrupted when the debugger is invoked. You can cause a different task to become the active task by using the debugger SET TASK/ACTIVE command (see Section 7.5).

You can specify the active task in debugger commands using the pseudotask name %ACTIVE_TASK. For example, the following command places the active task on HOLD:

```
DBG> SET TASK/HOLD %ACTIVE_TASK
```

The following command triggers a breakpoint at line 25 only when line 25 is executed by the task named CHILD:

```
DBG> SET BREAK %LINE 25 WHEN (%ACTIVE_TASK=CHILD)
```

7.2.3.2 Visible Task (%VISIBLE_TASK)

The *visible task* is the task whose stack and register set are the current context for looking up names, calls, and so on. In the following example, the value of the variable KEEP_COUNT in the context of the visible task is returned:

```
DBG> EXAMINE KEEP_COUNT
```

Initially, the visible task is the active task, but in a multitasking program, it may not always be the active task. You can cause a task to become the visible task by using the debugger SET TASK/VISIBLE command. However, making a task the visible task does not make it the active task.

You can specify the visible task in debugger commands with the pseudotask name %VISIBLE_TASK. For example, the following command obtains the task ID of the visible task:

```
DBG> EVALUATE %VISIBLE_TASK
```

The visible task is the task recognized by many of the debugger commands. In particular, the SET TASK command and its various qualifiers operate on the visible task; see Section 7.5.

7.2.3.3 Next Task (%NEXT_TASK)

The *next task* is the task that will execute when the visible task has finished executing. You can specify the next task in debugger commands using the pseudotask name %NEXT_TASK. The ordering of tasks is arbitrary but consistent within a single run of a program.

The pseudotask name %NEXT_TASK is useful for cycling through the total set of tasks that currently exist. For example, the following sequence of commands eventually cycles back to the task you started with:

```
DBG> SHOW TASK %VISIBLE_TASK; SET TASK/VISIBLE %NEXT_TASK
DBG> SHOW TASK %VISIBLE_TASK; SET TASK/VISIBLE %NEXT_TASK
   .
   .
   .
```

7.2.3.4 Caller Task (%CALLER_TASK)

The *caller task* is the task that called the entry associated with an accept statement, when the sequence of statements in the accept statement is being executed. You can specify the caller task in debugger commands using the pseudotask name %CALLER_TASK. This pseudotask name evaluates to the task ID of the task that called the entry associated with the accept statement. Otherwise, %CALLER_TASK evaluates to %TASK 0 (the null task). For example, %CALLER_TASK evaluates to %TASK 0 if the active task is not currently executing the accept statement.

For example, the following command sets a breakpoint within an accept statement of the sample program in Example 7–1:

```
DBG> SET BREAK %LINE 48
```

The accept statement in this case is being executed by task FATHER (%TASK 2) in response to a call of entry RENDEZVOUS by the main program (%TASK 1). Thus, when an EVALUATE %CALLER_TASK command is entered at this point, the result is the task ID of the calling task, the main program:

```
DBG> EVALUATE %CALLER_TASK
%TASK 1
DBG>
```

When the rendezvous is the result of an AST entry call, %CALLER_TASK evaluates to %TASK 0 because the caller is not a task. See the *VAX Ada Run-Time Reference Manual* for information on AST entry calls; see Section 7.2.2 for a definition of %TASK 0 (the null task).

7.2.4 Debugger Support of Ada Task Attributes

The Ada language defines the following attributes specific to tasks: T'CALLABLE, E'COUNT, T'STORAGE_SIZE, and T'TERMINATED, where T is a task type and E is a task entry (see the *VAX Ada Language Reference Manual* for more information on these attributes).

The debugger does not support these attributes, so you cannot enter commands such as EVALUATE CHILD'CALLABLE. However, you can obtain the information provided by each of these attributes with the debugger SHOW TASK command. See Section 7.3 for more information on this command.

7.3 Displaying Task Information (SHOW TASK)

You use the debugger SHOW TASK command to display information about one or more tasks in a multitasking program. The command format is as follows:

```
SHOW TASK[/qualifier[...]] [task-expression[,...]]
```

The SHOW TASK command has two kinds of qualifiers: *task-selection qualifiers*, which allow you to select tasks satisfying certain criteria; and *information qualifiers*, which provide additional information about specified tasks. Task expressions are defined in Section 7.2.

The following sections explain how to use the SHOW TASK command and its qualifiers.

7.3.1 Displaying Basic Information on All Tasks

The debugger SHOW TASK/ALL command provides basic information on all the tasks of a program that are currently in existence—namely, tasks that have been created and whose master has not yet terminated. For example:

```
DBG> SHOW TASK/ALL
  ❶       ❷  ❸    ❹       ❺            ❻
  task id pri hold state   substate   task object
* %TASK 1  7       RUN               SHARE$ADARTL+130428
  %TASK 2  7  HOLD SUSP  Accept      TASK_EXAMPLE.MOTHER+4
  %TASK 4  7       SUSP  Entry call TASK_EXAMPLE.FATHER_TYPE$TASK_BODY.CHILD+4
  %TASK 3  6       READY             TASK_EXAMPLE.MOTHER
DBG>
```

The information in each column is as follows:

❶ The task ID of the task. An asterisk indicates that the task is a visible task.

❷ The task priority. VAX Ada priorities range from 0 to 15. A task is created with a default priority of 7, unless another value is specified with the pragma PRIORITY.

❸ Indicates whether the task has been placed on HOLD with a SET TASK/HOLD command. Placing a task on HOLD restricts the state transitions it can make once the program is subsequently allowed to execute. A task placed on HOLD may enter any state except the RUNNING state (however, you can force it into the RUNNING state by using the SET TASK/ACTIVE command).

❹ Indicates the state of the task when the debugger interrupted program execution. The four possible states recognized by the debugger are identified in Table 7–1. Figure 7–1 shows the possible transitions of a task's state during program execution. Note from the SHOW TASK display that the states of Table 7–1 are abbreviated to RUN, READY, SUSP, and TERM, respectively.

❺ Indicates the substate of a task when the debugger interrupted program execution. The possible task substates refer to Ada-specific task conditions as identified in Table 7–2. The substate helps indicate the possible cause of a task's state. For example, if the current state of the task is SUSPENDED, then the entry in the substate column indicates the reason.

❻ A debugger path name for the task object, or the address of the task object if the debugger cannot determine its path name.

If you are debugging in screen mode, the following command causes changes in the SHOW TASK display (such as switches in task states) to be highlighted in reverse video:

```
DBG> SET DISPLAY/MARK_CHANGE T AT Q2 DO (SHOW TASK/ALL)
```

Here, T is the display name; Q2 specifies window Q2, which occupies the second quarter of the screen.

Note that you will receive an error message if you enter the SET DISPLAY/MARK_CHANGE command before the program has been elaborated (before typing GO to get to the beginning of the main program). Also, note that display T is updated only when the debugger gains control for some reason, such as at a breakpoint.

Table 7–1: Task States

Task State	Meaning
RUNNING	Currently running on the processor. This is the active task.
READY	Eligible to execute and waiting for the processor to be made available.
SUSPENDED	Suspended—that is, waiting for an event rather than for the availability of the processor. For example, when a task is created, it remains in the suspended state until it is activated.
TERMINATED	Terminated.

Figure 7–1: Task State Transitions

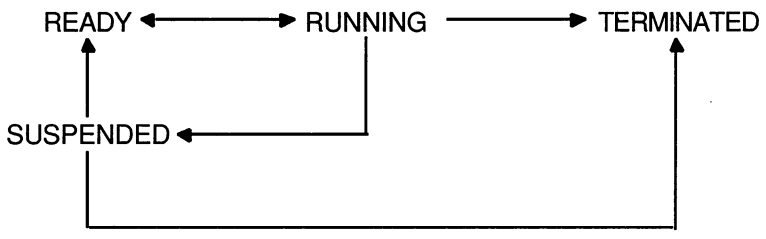


Table 7–2: Task Substates

Task Substate	Meaning
Abnormal	Task has been aborted.
Accept	Task is waiting at an accept statement that is not inside a select statement.
Activating	Task is elaborating its declarative part.
Activating tasks	Task is waiting for tasks it has created to finish activating.
Completed [abn]	Task is completed due to an abort statement, but is not yet terminated. In Ada, a *completed* task is one that is waiting for dependent tasks at its end statement. After the dependent tasks are terminated, the state changes to terminated.
Completed [exc]	Task is completed due to an unhandled exception,[1] but is not yet terminated. In Ada, a *completed* task is one that is waiting for dependent tasks at its end statement. After the dependent tasks are terminated, the state changes to terminated.
Completed	Task is completed. No abort statement was issued, and no unhandled exception[1] occurred.

[1]An unhandled exception is one for which there is no handler, or for which there is a handler that executes a raise statement and propagates the exception to an outer scope.

(continued on next page)

Table 7–2 (Cont.): Task Substates

Task Substate	Meaning
Delay	Task is waiting at a delay statement.
Dependents	Task is waiting for dependent tasks to terminate.
Dependents [exc]	Task is waiting for dependent tasks to allow an unhandled exception[1] to propagate.
Entry call	Task is waiting for its entry call to be accepted.
Invalid state	There is an error in the VAX Ada run-time library.
I/O or AST	Task is waiting for input-output completion or some AST.
Not yet activated	Task is waiting to be activated by the task that created it.
Select or delay	Task is waiting at a select statement with a delay alternative.
Select or terminate	Task is waiting at a select statement with a terminate alternative.
Select	Task is waiting at a select statement with no else, delay, or terminate alternative.
Shared resource	Task is waiting for an internal shared resource.
Terminated [abn]	Task was terminated by an abort statement.
Terminated [exc]	Task was terminated because of an unhandled exception.[1]
Terminated	Task terminated normally.
Timed entry call	Task is waiting in a timed entry call.

[1]An unhandled exception is one for which there is no handler, or for which there is a handler that executes a raise statement and propagates the exception to an outer scope.

7.3.2 Selecting Tasks for Display

You can select tasks for display with the debugger SHOW TASK command by specifying any of the following:

- A task list (a list of task expressions)
- Task selection qualifiers
- Both a task list and task selection qualifiers

If no parameters or task selection qualifiers are given, the SHOW TASK command displays summary information about the visible task.

The following sections discuss task lists and task selection qualifiers in more detail.

7.3.2.1 Task List

You specify a task list of one or more tasks with a series of task expressions separated by commas. For example, the following command selects the active task, %TASK 3, and task MOTHER for display:

```
DBG> SHOW TASK %ACTIVE_TASK,%TASK 3,MOTHER
```

Task expressions are defined in Section 7.2.

7.3.2.2 Task-Selection Qualifiers

You can use the task selection qualifiers listed in Table 7–3 with the debugger SHOW TASK command to select any tasks that satisfy all of a specified set of criteria. For example, the following command selects all tasks with priority 6:

```
DBG> SHOW TASK/PRIORITY=6
```

The following command selects all tasks that are either running or suspended:

```
DBG> SHOW TASK/STATE=(RUNNING,SUSPENDED)
```

When two or more task-selection qualifiers are used in the same SHOW TASK command, only those tasks that satisfy all specified criteria are selected for display. For example, the following command selects all tasks that are suspended and not on hold:

```
DBG> SHOW TASK/STATE=SUSPENDED/NOHOLD
```

Table 7–3: SHOW TASK Command Qualifiers for Task Selection

Qualifier	Meaning
/ALL	Selects all tasks that currently exist in the program for display. When you specify /ALL, you cannot specify a task list.
/HOLD	If you do not specify a task list, selects all tasks that are on HOLD. If you specify a task list, selects the tasks in the task list that are on HOLD.

(continued on next page)

Table 7–3 (Cont.): SHOW TASK Command Qualifiers for Task Selection

Qualifier	Meaning
/NOHOLD	If you do not specify a task list, selects all tasks that are not on HOLD. If you specify a task list, selects the tasks in the task list that are not on HOLD.
/PRIORITY=(n[,...])	If you do not specify a task list, selects all tasks that have any of the specified priorities, n, where n is a decimal integer from 0 to 15 inclusive. If you specify a task list, selects the tasks in the task list that have any of the priorities specified.
/STATE=(state[,...])	If you do not specify a task list, selects all tasks that are in any of the specified states (the possible states are RUNNING, READY, SUSPENDED, or TERMINATED). If you specify a task list, selects the tasks in the task list that are in any of the states specified.

7.3.2.3 Task List and Task Selection Qualifiers

When you specify both a task list and multiple task-selection qualifiers with the debugger SHOW TASK command, only the tasks that satisfy all specified criteria are selected for display. For example, the following command selects those tasks among the visible task, %TASK 3, and MOTHER that are in the RUNNING or SUSPENDED STATE, and have priority 7:

```
DBG> SHOW TASK/STATE=(RUN,SUSP)/PRIORITY=7 %VISIBLE_TASK, -
_DBG> %TASK 3,MOTHER
```

7.3.3 Obtaining Additional Information

You can use the information-selection qualifiers listed in Table 7–4 with the debugger SHOW TASK command to obtain specific information about all of the tasks in your program. You can use the information-selection qualifiers in conjunction with the task-selection techniques described in Sections 7.3.2.1 through 7.3.2.3.

Table 7–4: SHOW TASK Command Qualifiers for Information Selection

Qualifier	Meaning
/CALLS[=n]	Performs a SHOW CALLS command for each task selected for display (see Chapter 6 for a description of the SHOW CALLS command). You can use the SHOW CALLS command to obtain the current PC (program counter) of a task.
/FULL	Displays additional information about each task selected for display. /FULL provides additional information if used either by itself, or with the /CALLS or /STATISTICS qualifier.
/STATISTICS	Displays tasking statistics for the entire tasking system. When you specify /STATISTICS, the only other permissible qualifier is /FULL.
/TIME_SLICE	Displays the current value of the pragma TIME_SLICE.

The SHOW TASK/FULL command provides detailed information about each task selected for display. For example:

```
❶  task id     pri hold state   substate          task object
  * %TASK 2      7       RUN                    TASK_EXAMPLE.MOTHER+4

❷       Waiting entry callers:
          Waiters for entry BOGUS:
              %TASK 4, type: CHILD

❸      Task type:       FATHER_TYPE
       Created at PC:   TASK_EXAMPLE.%LINE 14+22
       Parent task:     %TASK 1
       Start PC:        TASK_EXAMPLE.FATHER_TYPE$TASK_BODY
❹      Task control block:          ❺ Stack storage (bytes):
         Task value:    490816            RESERVED_BYTES:     10640
         Entries:       3                 TOP_GUARD_SIZE:      5120
         Size:          1488              STORAGE_SIZE:       30720
❻      Stack addresses:                 Bytes in use:         456
         Top address:   001EB600
         Base address:  001F2DFC       ❼ Total storage:      47968
```

The following notes are keyed to this example:

❶ Identifying information about the task.

❷ Rendezvous information. If the task is a caller task, lists the entries for which it is queued. If the task is to be called, gives information about the kind of rendezvous that will take place and lists the callers that are currently queued for any of the task's entries.

❸ Task context information.

❹ Task control block information. The task value is the address, in decimal notation, of the task control block.

❺ Stack storage information:

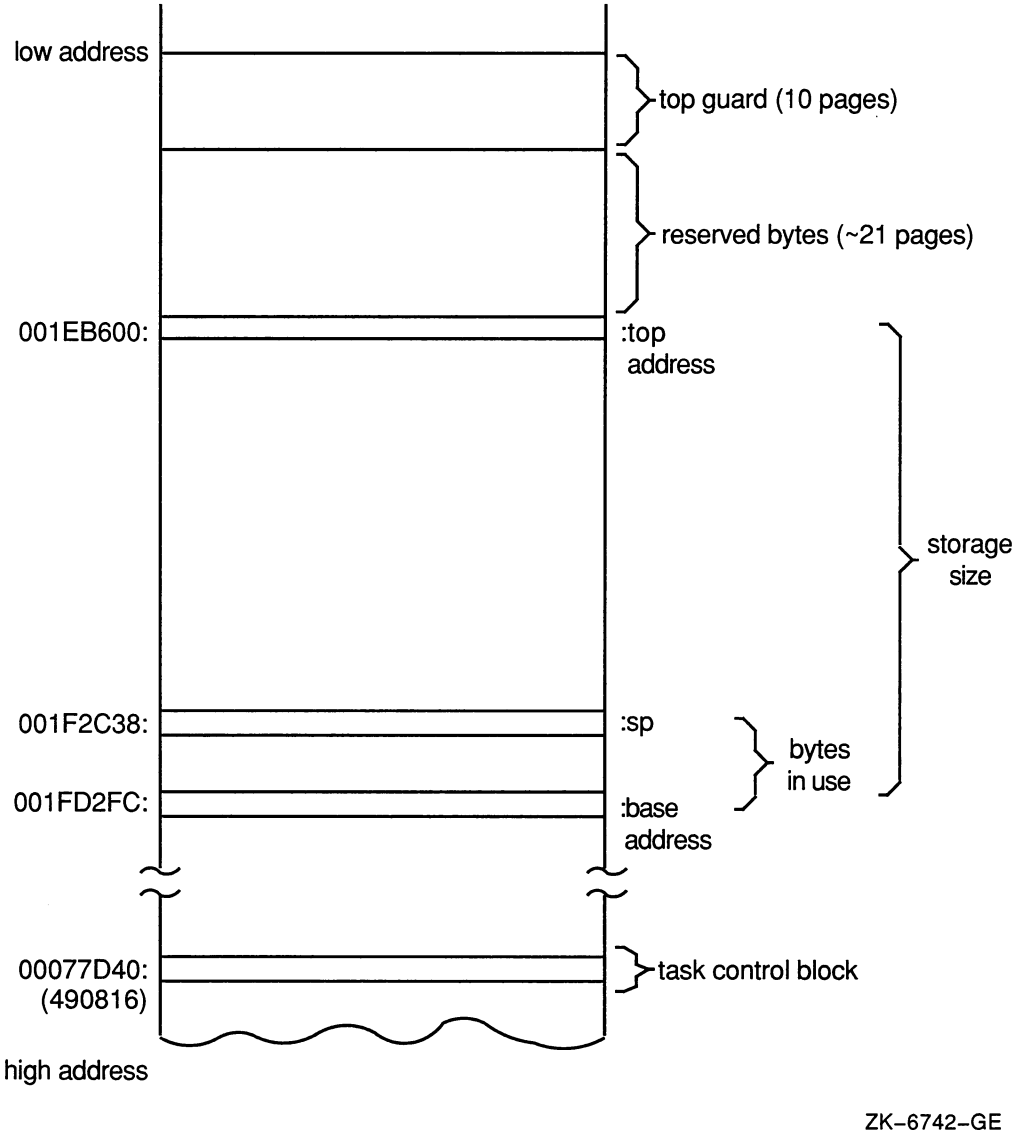
- RESERVED_BYTES gives the storage allocated by the Ada run-time library for handling stack overflow.
- TOP_GUARD_SIZE gives the storage allocated for guard pages, which provide protection against storage overflow during task execution. You can specify the number of bytes to be allocated as guard pages with the VAX Ada pragmas TASK_STORAGE and MAIN_STORAGE; the number shown by the debugger is the number of bytes allocated (the pragma value is rounded up to an integral number of pages, as necessary). See the *VAX Ada Language Reference Manual* and *VAX Ada Run-Time Reference Manual* for more information about these pragmas and the top guard storage area.
- STORAGE_SIZE gives the storage allocated for the task activation. You can specify the number of bytes to be allocated with the T'STORAGE_SIZE representation clause or in the VAX Ada pragma MAIN_STORAGE; the number shown by the debugger is the number of bytes allocated (the value specified is rounded up to an integral number of pages, as necessary). See the *VAX Ada Language Reference Manual* and *VAX Ada Run-Time Reference Manual* for more information about this representation clause and pragma and about the task activation (working) storage area.
- "Bytes in use:" gives the size of the task stack.

❻ Stack addresses of the task stack.

❼ The total storage used by the task. Adds together the task control block size, the number of reserved bytes, the top guard size, and the storage size.

Figure 7–2 shows the task stack for task FATHER.

Figure 7–2: Diagram of a Task Stack



The SHOW TASK/STATISTICS command reports some statistics about all of the tasks in your program. The SHOW TASK/STATISTICS/FULL command reports more of them. For example:

```
DBG> SHOW TASK/STATISTICS/FULL

task statistics
    Entry calls       = 4       Accepts = 1     Selects = 2
    Tasks activated   = 3       Tasks terminated   = 0
    ASTs delivered    = 4       Hibernations       = 0
    Total schedulings = 15
        Due to readying a higher priority task = 1
        Due to task activations                = 3
        Due to suspended entry calls           = 4
        Due to suspended accepts               = 1
        Due to suspended selects               = 2
        Due to waiting for a DELAY             = 0
        Due to scope exit awaiting dependents  = 0
        Due to exception awaiting dependents   = 0
        Due to waiting for I/O to complete     = 0
        Due to delivery of an AST              = 4
        Due to task terminations               = 0
        Due to shared resource lock contention = 0
```

You can use this statistics information to measure the performance of your tasking program. The larger the number of total schedulings (also known as context switches), the more tasking overhead there is.

7.4 Examining and Manipulating Tasks

The debugger EXAMINE command (or EXAMINE/TASK command), applied to a task object, displays the task ID. For example:

```
DBG> EXAMINE FATHER
TASK_EXAMPLE.FATHER:    %TASK 2
DBG>
```

You can use the EXAMINE/HEXADECIMAL command (or the EXAMINE/TASK/HEXADECIMAL command) to determine the 8-digit hexadecimal task value. (In VAX Ada, the task value is the address of the task control block of a specified task.) For example:

```
DBG> EXAMINE/HEXADECIMAL FATHER
TASK_EXAMPLE.FATHER: 0015AD00
DBG>
```

7.5 Changing Task Characteristics (SET TASK)

You use the debugger SET TASK command to change a task's characteristics as you debug your program. The command format is as follows:

```
SET TASK[/qualifier[...]] [task-expression[,...]]
```

Table 7–5 defines the SET TASK command qualifiers. Section 7.2 defines task expressions. Note that if no qualifier is specified, the /VISIBLE qualifier is assumed by default.

Table 7–5: SET TASK Command Qualifiers

Qualifier	Meaning
	Task Selection Qualifiers
/ALL	Applies the SET TASK command to all tasks. When you specify /ALL, you cannot specify a task list, nor can you specify the /ACTIVE, /VISIBLE, or /TIME_SLICE qualifiers.
	Attribute Qualifiers
/ABORT	Aborts the specified tasks. If no task list is specified, aborts the visible task. Note that the task is marked for termination but is not immediately terminated. The effect is identical to executing the Ada statement **abort** task-name, and causes the specified tasks to become abnormal.
/ACTIVE	Makes the specified task the active task. Causes a task switch to the new active task and resets the visible task to be the new active task. The specified task must be in either the RUNNING or READY state. You must specify only one task.

(continued on next page)

Table 7–5 (Cont.): SET TASK Command Qualifiers

Qualifier	Meaning
	Attribute Qualifiers
/HOLD	Places the specified tasks on HOLD. If no task list is specified, places the visible task on HOLD. Placing a task on HOLD prevents a task from entering the RUNNING state. A task placed on HOLD is allowed to make other state transitions; in particular, it may change from the SUSPENDED to the READY state. A task that is already in the RUNNING state (the active task) can continue to execute as long as it remains in the RUNNING state, even though it is placed on HOLD. If the task leaves the RUNNING state for any reason (including expiration of a time slice, if time slicing is enabled), it may not return to the RUNNING state until the HOLD is removed. You can force a task into the RUNNING state with the SET TASK/ACTIVE command even if the task is on HOLD.
/NOHOLD	Removes the specified tasks from HOLD. If no task list is specified, removes the visible task from HOLD.
/PRIORITY=n	Sets the priority of the specified tasks to n, where n is a decimal integer from 0 to 15, inclusive. If no task list is specified, sets the priority of the visible task to n. Note that this does not prevent the task's priority from later changing in the course of execution, for example, while executing a rendezvous.
/RESTORE	Causes the priority of the specified tasks to be restored to the value specified in a pragma PRIORITY. If a pragma PRIORITY was not specified, the default value of 7 is used. If no task list is specified, causes the priority of the visible task to be restored.
/TIME_SLICE=t	Sets the duration otherwise specified by the pragma TIME_SLICE to the value t, where t is a decimal integer or fixed-point value representing seconds (see Section 7.7.2). The SET TASK/TIME_SLICE=0.0 command disables time slicing.
/VISIBLE	Makes the specified task the visible task. You must specify only one task.

Most of the qualifiers provide a means of controlling the tasking environment by directly or indirectly causing task state transitions. In contrast, the /VISIBLE qualifier is used to direct subsequent debugger commands, such as EXAMINE, to an individual task. See Section 7.2.3.2 for more information on the visible task.

Task switching may be confusing when you are trying to debug a program. The SET TASK/TIME_SLICE and SET TASK/HOLD commands give you several ways of controlling task switching.

The SET TASK/HOLD/ALL command freezes the state of all tasks (except the active task). You can use this command in combination with the SET TASK/ACTIVE command to observe the behavior of one or more specified tasks in isolation, by executing the active task with the STEP or GO command, then switching execution to another task with the SET TASK/ACTIVE command. For example:

```
DBG> SET TASK/HOLD/ALL
DBG> SET TASK/ACTIVE %TASK 1
DBG> GO
   .
   .
   .
DBG> SET TASK/ACTIVE %TASK 3
DBG> STEP
   .
   .
   .
```

7.6 Setting Breakpoints and Tracepoints

You can use the debugger SET BREAK and SET TRACE commands with tasking programs just as you use them with nontasking programs. You can also take advantage of the following task-related features:

- Task-specific and task-independent debugger eventpoints
- Task body, entry call, and accept statement breakpoints and tracepoints
- The /EVENT=event-name qualifier (which allows you to set a breakpoint or tracepoint when a task makes a state transition)

The following sections explain how to use these features.

7.6.1 Task-Specific and Task-Independent Debugger Eventpoints

An eventpoint is an event that you can use to return control to the debugger. An eventpoint is set by a debugger command to instruct the debugger to watch for the specified event, and is triggered when the debugger observes the event. Breakpoints, tracepoints, watchpoints, and step commands are eventpoints.

Task-independent eventpoints can be triggered by the execution of any task in a program, regardless of which task is active when the eventpoint is set. Task-independent eventpoints are generally specified by an address expression such as a line number or a name. All watchpoints are task-independent eventpoints. For example:

```
DBG> SET BREAK COUNTER
DBG> SET BREAK/NOSOURCE %LINE 42, CHILD$TASK_BODY
DBG> SET WATCH/AFTER=3 KEEP_COUNT
```

A *task-specific eventpoint* can be set only for the task that is active when the command is entered. A task-specific eventpoint is triggered only when that same task is active. For example, the STEP/LINE command is a task-specific eventpoint: other tasks may execute the same Ada source line and not trigger the event.

The following eventpoints are task specific. Any other eventpoints, including all those set with the SET WATCH command, are task independent.

STEP/BRANCH
STEP/CALL
STEP/INSTRUCTION[=opcode]
STEP/LINE
STEP/RETURN

SET BREAK/BRANCH
SET BREAK/CALL
SET BREAK/INSTRUCTION[=opcode]
SET BREAK/LINE

SET TRACE/BRANCH
SET TRACE/CALL
SET TRACE/INSTRUCTION[=opcode]
SET TRACE/LINE

For example, the following eventpoints are task specific:

```
DBG> SET BREAK/INSTRUCTION
DBG> SET TRACE/INSTRUCTION/SILENT DO (EXAMINE KEEP_COUNT)
DBG> STEP/CALL/NOSOURCE
```

To work around this restriction, you can use a WHEN clause. For example:

```
DBG> SET BREAK %LINE 10 WHEN (%ACTIVE_TASK=FATHER)
```

7.6.2 Task Bodies, Entry Calls, and Accept Statements

You can always use line numbers when setting breakpoints or tracepoints. However, names, if they exist, are preferable as address expressions because they are more stable as you modify your program.

As discussed in Section 7.2.1, you can use one of the following two forms when referring to a task body in a debugger command:

```
task-type-identifier$TASK_BODY
task-identifier$TASK_BODY
```

For example, the following command sets a breakpoint on the body of task CHILD. This breakpoint is triggered just before the elaboration of the task's declarative part (also called the task's activation) :

```
DBG> SET BREAK CHILD$TASK_BODY
DBG>
```

Note that CHILD$TASK_BODY is a name for the address of the first instruction the task will execute. It is meaningful to set a breakpoint on an instruction, and hence on this name. However, you must not name the task object (for example, CHILD) in a SET BREAK command. The task-object name designates the address of a data item (the 32-bit task value). Just as it is erroneous to set a breakpoint on an integer object, it is erroneous to set a breakpoint on a task object.

You can monitor the execution of communicating tasks by setting breakpoints or tracepoints on entry calls and accept statements. There are several points in and around an accept statement where you may want to set a breakpoint or tracepoint. For example, consider the following program segment, which has two accept statements for the same entry, RENDEZVOUS:

```
 8   task body TWO_ACCEPTS is
 9   begin
10      for I in 1..2 loop
11         select
12            accept RENDEZVOUS do
13               PUT_LINE("This is the first accept statement");
14            end RENDEZVOUS;
15         or
16            terminate;
17         end select;
18      end loop;
19      accept RENDEZVOUS do
20         PUT_LINE("This is the second accept statement");
21      end RENDEZVOUS;
22   end TWO_ACCEPTS;
```

You can set a breakpoint or tracepoint in the following places in this example:

1. At the start of an accept statement (line 12 or 19). By setting a breakpoint or tracepoint here, you can monitor when execution reaches the start of the accept statement, where the accepting task may become suspended before a rendezvous actually occurs.
2. At the start of the body (sequence of statements) of an accept statement (line 13 or 20). By setting a breakpoint or tracepoint here, you can monitor when a rendezvous has been initiated—that is, when the accept statement actually begins execution.
3. At the end of an accept statement (line 14 or 21). By setting a breakpoint or tracepoint here, you can monitor when the rendezvous has completed, and execution is about to switch back to the caller task.

To set a breakpoint or tracepoint in and around an accept statement, you can specify the associated line number. For example, the following command sets a breakpoint on the start and also on the body of the first accept statement in the preceding example:

```
DBG> SET BREAK %LINE 12, %LINE 13
```

To set a breakpoint or a tracepoint on an accept statement body, you can also use the entry name (specifying its expanded name to identify the task body where the entry is declared). For example:

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS
```

If there is more than one accept statement for an entry, the debugger treats the entry as an overloaded name. In other words, the debugger issues a message indicating that the symbol is overloaded, and you must use the SHOW SYMBOL command to identify the overloaded names that have been assigned by the debugger. For example:

```
DBG> SHOW SYMBOL RENDEZVOUS
overloaded symbol TEST.TWO_ACCEPTS$TASK_BODY.RENDEZVOUS
  overloaded instance TEST.TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__1
  overloaded instance TEST.TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2
```

Note that overloaded names have an integer suffix preceded by two underscores; see Chapter 6 for more information on overloaded names.

You can use the EXAMINE/SOURCE command to determine which name is associated with a particular accept statement. For example:

```
DBG> EXAMINE/SOURCE TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__1
module TEST_ACCEPTS
    12:       accept RENDEZVOUS do
DBG> EXAMINE/SOURCE TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2
module TEST_ACCEPTS
    19:       accept RENDEZVOUS do
```

In the following example, when the breakpoint is triggered, the caller task is evaluated:

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2 -
_DBG> DO (EVALUATE %CALLER_TASK)
```

You can cause a breakpoint to trigger only under some circumstances. For example, the following command triggers a breakpoint only when the calling task is %TASK 2:

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2 -
_DBG> (WHEN (%CALLER_TASK = %TASK 2))
```

If the calling task has more than one entry call to the same accept statement, you can use the SHOW TASK/CALLS command to identify the source line where the entry call was issued. For example:

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2 -
_DBG> DO (SHOW TASK/CALLS %CALLER_TASK)
```

7.6.3 Monitoring Ada Task Events

The debugger SET BREAK and SET TRACE commands each have an /EVENT=event-name qualifier. You can use this qualifier to set breakpoints or tracepoints that will be triggered by Ada exception and tasking events; the tasking events are discussed in this section (see Chapter 6 for more information on the exception events). When a breakpoint or tracepoint is triggered as a result of an event name qualifier, the debugger identifies the Ada event that caused it to be triggered and gives additional information.

The general command syntax for the SET BREAK/EVENT=event-name command is as follows (see Chapter 6 for more information on setting breakpoints and tracepoints; see the *VMS Debugger Manual* for more information on debugger syntax):

```
SET BREAK/EVENT=event-name [task-expr[,...]]
SET TRACE/EVENT=event-name [task-expr[,...]]
```

The events specified with the /EVENT=event-name qualifier are language dependent. When you run a program under debugger control, the appropriate set of events is defined during the initialization of language-specific parameters. (The SET EVENT_FACILITY command allows you to initialize the debugger for events pertinent to any language.)

Table 7–6 defines the set of events (event-name keyword values) that apply to VAX Ada (the exception-related events are included for completeness). You can obtain a list of these events from the debugger by entering the SHOW EVENT_FACILITY command, which also identifies the currently set event facility.

You can abbreviate an event name to the minimum number of characters that make it unique.

Table 7–6: VAX Ada Event Names

Event Name	Description
Exception-Related Events	
HANDLED	Triggers when an exception is about to be handled in some Ada exception handler, including an **others** handler (see Chapter 6).
HANDLED_OTHERS	Triggers only when an exception is about to be handled in an **others** Ada exception handler (see Chapter 6).
Task Exception-Related Events	
RENDEZVOUS_EXCEPTION	Triggers when an exception begins to propagate out of a rendezvous.
DEPENDENTS_EXCEPTION	Triggers when an exception causes a task to wait for dependent tasks in some scope (includes unhandled exceptions,[1] which, in turn, include special exceptions internal to the VAX Ada run-time library; see the *VAX Ada Run-Time Reference Manual* for more information). Often immediately precedes a deadlock.

[1]An unhandled exception is an exception that either has no handler in the current frame, or that has a handler which executes a raise statement and propagates the exception to an outer scope.

(continued on next page)

Table 7–6 (Cont.): VAX Ada Event Names

Event Name	Description
Task Termination Events	
TERMINATED	Triggers when a task is terminating, whether normally, by an abort statement, or by an exception.
EXCEPTION_TERMINATED	Triggers when a task is terminating due to an unhandled exception.[1]
ABORT_TERMINATED	Triggers when a task is terminating due to an abort statement.
Low-Level Task Scheduling Events	
RUN	Triggers when a task is about to run.
PREEMPTED	Triggers when a task is being preempted from the RUN state and its state changes to READY. (See Figure 7–1.)
ACTIVATING	Triggers when a task is about to begin its activation (that is, at the beginning of the elaboration of the declarative part of its task body).
SUSPENDED	Triggers when a task is about to be suspended.

[1]An unhandled exception is an exception that either has no handler in the current frame, or that has a handler which executes a raise statement and propagates the exception to an outer scope.

The following examples show the use of the /EVENT=event-name qualifier.

```
DBG> SET TRACE/EVENT=RUN CHILD,%TASK 2
```

This command sets tracepoints on the tasks CHILD and %TASK 2. Each tracepoint is triggered whenever its associated task makes a transition to the RUN state.

The next command sets a breakpoint that is triggered whenever a task enters the TERMINATED state. A SHOW TASK/ALL command is entered at each breakpoint:

```
DBG> SET BREAK/EVENT=TERMINATED DO (SHOW TASK/ALL)
```

Breakpoints for the EXCEPTION_TERMINATED and DEPENDENTS_EXCEPTION events are automatically set for you when you invoke the debugger with a VAX Ada program (or with a program in a supported language that is linked with a VAX Ada compilation unit). You can see that these breakpoints are set when you enter a SHOW BREAK command.

The EXCEPTION_TERMINATED event triggers when a task is being terminated because of an exception. That condition usually indicates an unanticipated program error. In the following example, the SET BREAK command is shown only for emphasis, as the debugger automatically breaks on EXCEPTION_TERMINATED events:

```
DBG> SET BREAK/EVENT=EXCEPTION_TERMINATED
DBG> GO
    .
    .
    .

break on ADA event EXCEPTION_TERMINATED
  Task %TASK 2 is terminating because of an exception
    %ADA-F-EXCCOP, Exception was copied at a "raise;" or "accept"
    -ADA-F-EXCEPTION, Exception SOME_ERROR
    -ADA-F-EXCRAIPRI, Exception raised prior to PC = 00000B61

DBG>
```

The DEPENDENTS_EXCEPTION event often unexpectedly precedes a deadlock. For example (again, the SET BREAK command is shown only for emphasis):

```
DBG> SET BREAK/EVENT=DEPENDENTS_EXCEPTION
DBG> GO
    .
    .
    .

break on ADA event DEPENDENTS_EXCEPTION
  Task %TASK 2 may await dependent tasks because of this exception:
    %ADA-F-EXCCOP, Exception was copied at a "raise;" or "accept"
    -ADA-F-EXCEPTION, Exception SOME_ERROR
    -ADA-F-EXCRAIPRI, Exception raised prior to PC = 00000B61

DBG>
```

The RENDEZVOUS_EXCEPTION event allows you to see an exception before it leaves a rendezvous (before exception information has been lost due to copying the exception into the calling task). For example:

```
DBG> SET BREAK/EVENT=RENDEZVOUS_EXCEPTION
DBG> GO
    .
    .
    .
break on ADA event RENDEZVOUS_EXCEPTION
  Exception is propagating out of a rendezvous in task %TASK 2
    %ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
    -ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000BA6

DBG>
```

You can use the SHOW BREAK and SHOW TRACE commands to identify the event breakpoints or tracepoints that are currently set.

To cancel breakpoints or tracepoints set with the /EVENT=event-name qualifier, you use the CANCEL BREAK/EVENT=event-name or CANCEL TRACE/EVENT=event-name command, respectively.

The CANCEL BREAK/EVENT=event-name (or TRACE) command cancels a breakpoint (or tracepoint) set by the SET BREAK/EVENT=event-name (or TRACE) command. To cancel a breakpoint or tracepoint associated with an event name, you must specify the event qualifier and optional task expression in the CANCEL command exactly as you did with the SET command, excluding any WHEN and DO clauses. For example, if you enter the CANCEL BREAK/EVENT=TERMINATED command without a parameter, it will not cancel a breakpoint that was set with a parameter; it will cancel only a breakpoint that was set with the SET BREAK/EVENT=TERMINATED command, with no parameter specified.

You may want to set certain event breakpoints and tracepoints in a debugger initialization file for tasking programs (the general use of initialization files is explained in Chapter 6). The sample initialization file in Example 7–2 may be useful in helping you to locate task-related errors.

Example 7–2: Sample Debugger Initialization File for VAX Ada Tasking Programs

```
SET OUTPUT VERIFY
SET OUTPUT LOG
!
SET BREAK/EVENT=ACTIVATING
!  Break on any task activations
!
SET BREAK/EVENT=HANDLED DO (SHOW CALLS)
!  Traceback on any exception handling
!
SET BREAK/EVENT=HANDLED_OTHERS DO (SHOW CALLS)
!  Traceback on any 'when others' handlers
!
SET BREAK/EVENT=DEPENDENTS_EXCEPTION DO (SHOW CALLS)
!  Traceback on any exceptions awaiting the termination
!  of dependent tasks
!
SET BREAK/EVENT=RENDEZVOUS_EXCEPTION
!  Break on any rendezvous involving exceptions
!
SET BREAK/EVENT=ABORT_TERMINATED DO (SHOW CALLS)
!  Traceback on all task terminations caused by
!  abort statements
!
SET BREAK/EVENT=EXCEPTION_TERM DO (SHOW CALLS)
!  Traceback on any task terminations caused by
!  unhandled exceptions
!
SET BREAK/EVENT=TERMINATED
!  Break on any task terminations
!
DEFINE/COMMAND sta="SHOW TASK/ALL"
!  Define a shorter command for displaying task statistics
!
DEFINE/COMMAND stf="SHOW TASK/FULL"
!  Define a shorter command for displaying full
!  information about one or more particular tasks
!
DEFINE/COMMAND noslice="SET TASK/TIME=0.0"
!  Define a shorter command for disabling time slicing
!
DEFINE/COMMAND slice="SET TASK/TIME="
!  Define a shorter command for enabling time slicing
```

7.7 Additional Task-Debugging Topics

The following sections discuss additional topics related to task debugging:

- Deadlock
- Time slicing
- Using CTRL/Y
- Automatic stack checking
- Highlighting task state changes

7.7.1 Debugging Programs with Deadlock

Deadlock is an error condition in which each task in a group of tasks is suspended and no task in the group can resume execution until some other task in the group executes. Deadlock is a typical error in tasking programs (in much the same way that infinite loops are typical errors in programs that use while statements).

Deadlock is easy to detect: it causes your program to appear to suspend, or hang, in midexecution. When deadlock occurs in a program that is running under the control of the debugger, you must first press CTRL/Y to interrupt the deadlock. Then, after entering the DCL DEBUG command, you can resume debugging.

In general, the debugger command SHOW TASK/ALL or SHOW TASK/STATE=SUSPENDED is useful because it shows which tasks are suspended in your program and why. The SHOW TASK/FULL command is useful because it gives detailed task state information, including information about rendezvous, entry calls, and entry index values. The /EVENT=event-name qualifier is useful because it allows you to trace or set breakpoints at or near locations that may lead to deadlock. The SET TASK/PRIORITY and SET TASK/RESTORE commands are useful because they allow you to see if a low-priority task that never runs is causing the deadlock.

Table 7–7 lists a number of kinds of deadlock and suggests debugger commands that are useful in diagnosing the cause of the deadlock. Previous sections of this chapter describe each of the task debugging commands in detail.

Table 7–7: Kinds of Deadlock and Debugger Commands for Diagnosing Them

Kind of Deadlock	Debugger Commands
Self-calling deadlock (a task calls one of its own entries)	SHOW TASK/ALL, SHOW TASK/SUSPENDED, SHOW TASK/FULL
Circular-calling deadlock (a task calls another task, which calls the first task)	SHOW TASK/ALL, SHOW TASK/SUSPENDED, SHOW TASK/FULL
Dynamic-calling deadlock (a circular series of entry calls exists, and at least one of the calls is a timed or conditional entry call in a loop)	SHOW TASK/ALL, SHOW TASK/SUSPENDED, SHOW TASK/FULL
Exception-induced deadlock (an exception prevents a task from answering one of its entry calls, or the propagation of an exception must wait for dependent tasks)	SHOW TASK/ALL, SHOW TASK/SUSPENDED, SHOW TASK/FULL, SET BREAK/EVENT=DEPENDENTS_ EXCEPTION, SET TRACE/EVENT=DEPENDENTS_ EXCEPTION
Deadlock due to incorrect run-time calculations for entry indexes, **when** conditions, and delay statements within select statements	SHOW TASK/ALL, SHOW TASK/STATE=SUSPENDED, SHOW TASK/FULL, EXAMINE
Deadlock due to entries being called in the wrong order	SHOW TASK/ALL, SHOW TASK/STATE=SUSPENDED, SHOW TASK/FULL
Deadlock due to busy-waiting on a variable used as a flag that is to be set by a lower priority task, and the lower priority task never runs because a higher priority task is always ready to execute	SHOW TASK/ALL, SHOW TASK/STATE=SUSPENDED, SHOW TASK/FULL, SET TASK/PRIORITY, SET TASK/RESTORE

7.7.2 Debugging Programs that Use Time Slicing

Tasking programs that use time slicing (as specified in the program with the pragma TIME_SLICE) are difficult to debug because time slicing makes the relative behavior of tasks asynchronous. In other words, without time slicing, task execution is determined solely by task priority; task switches are predictable and the behavior of the program is repeatable from one run

to the next. With time slicing, task priorities still govern task switches, but tasks of the same priority also take turns executing for a specified period of time. Time slicing thus causes tasks to execute more independently from each other, and the behavior of a program that uses time slicing may not be repeatable from one run of the program to the next.

The debugger SET TASK/TIME_SLICE=t command allows you to disable time slicing (SET TASK/TIME_SLICE=0.0) or specify a new value for a pragma TIME_SLICE. Thus, you can use this command to tune the execution of your tasking programs, or to diagnose problems that may be masked by the use of time slicing.

Note that there is an interaction between VAX Ada's time slicing and the debugger watchpoint implementation. When you set watchpoints, the debugger may automatically increase the value of the pragma TIME_SLICE to 10.0. Slowing down the time-slice rate prevents these problems from occurring.

For more information on the effect of time slicing on task switching, see the *VAX Ada Run-Time Reference Manual*; for more information on the pragma TIME_SLICE, see the *VAX Ada Language Reference Manual.*

7.7.3 Using CTRL/Y when Debugging Tasks

You may experience some problems invoking the debugger with the DCL DEBUG command after interrupting a task debugging session with CTRL/Y. In such cases, you should insert the following two lines in the source code at the beginning of your main program to name the VAX Ada predefined package CONTROL_C_INTERCEPTION:

```
with CONTROL_C_INTERCEPTION;
pragma ELABORATE(CONTROL_C_INTERCEPTION);
```

Then, you should use CTRL/C instead of CTRL/Y to interrupt your task debugging session. See the *VAX Ada Run-Time Reference Manual* for information on this package.

7.7.4 Automatic Stack Checking in the Debugger

In tasking programs, an undetected stack overflow can occur in certain circumstances, and can lead to unpredictable execution (see the *VAX Ada Run-Time Reference Manual* for more information on task stack overflow). The debugger automatically performs the following stack checks to help you detect the source of stack overflow problems:

- If the stack pointer is out of bounds, the debugger displays an error message.
- A stack check is performed for the active task after a STEP or breakpoint eventpoint triggers (see Section 7.6.1). (This check is not performed if you have used the /SILENT qualifier with the STEP or SET BREAKPOINT command.)
- A stack check is performed for each task whose state is displayed in a SHOW TASK command. Thus, a SHOW TASK/ALL command automatically causes the stacks of all tasks to be checked.

The following examples show the kinds of error messages displayed by the debugger when a stack check fails. Note that a warning is issued when most of the stack has been used up, even if the stack has not yet overflowed.

```
warning: %TASK 2 has used up over 90% of its stack
  SP: 0011194C  Stack top at: 00111200  Remaining bytes: 1868

error: %TASK 2 has overflowed its stack
  SP: 0010E93C  Stack top at: 00111200  Remaining bytes: -10436

error: %TASK 2 has underflowed its stack
  SP: 7FF363A4  Stack base at: 001189FC  Stack top at: 00111200
```

One of the unpredictable events that can happen after a stack overflows is that the stack can then underflow. For example, if a task stack overflows and the stack pointer remains in the top guard area, the VMS operating system will attempt to signal an ACCVIO condition. However, because the top guard area is not a writable area of the stack, the VMS operating system cannot write the signal arguments for the ACCVIO. When this happens, the VMS operating system cuts back the stack: it causes the frame pointer and stack pointer to point to the base of the main program stack area, writes the signal arguments, and then modifies the program counter to force an image exit. If a time-slice AST or other AST occurs at this instant, execution can resume in a different task, and for a while, the program may continue to execute, although not normally (the task whose stack overflowed may use—and overwrite—the main program stack). The debugger stack checks help you to detect this situation. If you step into a task whose stack has been cut back by the VMS system, or if you use SHOW TASK/ALL at that time, the debugger will issue its stack underflow message.

Appendix A

ACS Command Dictionary

This appendix is a dictionary of all of the ACS commands, plus the DCL ADA command. The commands are organized alphabetically, with full descriptions of their format, parameters, and qualifiers, and with examples of their use. See Chapter 1 for general information on using ACS commands. See Chapter 2 for the conventions on specifying unit names.

In this appendix, qualifiers are categorized according to the VMS DCL qualifier conventions (see the *VMS DCL Concepts Manual*). In other words, a qualifier may belong to one of three types:

- A *command* qualifier has the same effect, regardless of where it appears in the command string (whether it is appended to the command verb or to a parameter).
- A *positional* qualifier has a different effect depending on where it appears in the command string. A positional qualifier appended to the command verb affects the entire command string. A positional qualifier appended to a parameter affects only that parameter.
- A *parameter* qualifier can be used only with a specified parameter. It cannot be appended to the command verb.

Qualifiers remain unique when truncated to their first four characters, not including the NO of the negative form. In command procedures, to guarantee compatibility with future releases of VAX Ada, you should not use fewer than four characters.

The examples in this appendix, as those throughout the manual, use the file-name conventions described in Chapter 1. Also, examples of messages issued by the compiler, program library manager, and so on display only the severity level and the message text. No facility name or message ID is shown.

($) ADA

Invokes the VAX Ada compiler to compile one or more VAX Ada source files.

NOTE

The ADA command is a DCL command, not an ACS command.

Format

ADA *file-spec[,...]*

Command Qualifiers	**Defaults**
/LIBRARY=directory-spec	/LIBRARY=ADA$LIB

Positional Qualifiers	**Defaults**
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/[NO]CHECK	See text.
/[NO]COPY_SOURCE	/COPY_SOURCE
/[NO]DEBUG[=(option[,...])]	/DEBUG=ALL
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]ERROR_LIMIT[=n]	/ERROR_LIMIT=30
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOAD[=option]	/LOAD=REPLACE
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]NOTE_SOURCE	/NOTE_SOURCE
/[NO]OPTIMIZE[=(option[,...])]	See text.
/[NO]SHOW[=option]	/SHOW=PORTABILITY
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/[NO]WARNINGS[=(option[,...])]	See text.

Prompts

_File:

($) ADA

Command Parameters

file-spec[,...]
Specifies one or more VAX Ada source files to be compiled. If you do not specify a file type, the compiler uses the default file type of .ADA. No wildcard characters are allowed in the file specifications.

If you specify more than one input file, you must separate the file specifications with commas (,). You cannot use plus signs (+) to separate file specifications.

Description

The DCL ADA command is one of four VAX Ada compilation commands. The other three compilation commands are the ACS LOAD, COMPILE, and RECOMPILE commands.

The ADA command can be used at any time to compile one or more source files (.ADA). VAX Ada source files are compiled in the order in which they appear in the command line. If a source file contains more than one VAX Ada compilation unit, the units are compiled in the order in which they appear in a source file. The Ada rules governing compilation order are summarized in Chapter 1.

The ADA command compiles units in the context of the current program library. Whenever a compilation unit is compiled without error, the current program library is updated with the object module and other products of compilation.

See Chapters 2 and 3 for more information on VAX Ada program libraries, sublibraries, and compilation.

Command Qualifiers

/LIBRARY=directory-spec
Specifies the program library that is to be the current program library for the duration of the compilation. The directory specified must be an existing VAX Ada program library. No wildcard characters are allowed in the directory specification.

By default, the current program library is the program library last specified in an ACS SET LIBRARY command. The logical name ADA$LIB is assigned to the program library specified in an ACS SET LIBRARY command.

Positional Qualifiers

/ANALYSIS_DATA[=file-spec]
/NOANALYSIS_DATA (D)
Controls whether a data analysis file containing source code cross-reference and static analysis information is created. The data analysis file is supported only for use with Digital layered products, such as the VAX Source Code Analyzer.

One data analysis file is created for each source file that is compiled. The default directory for data analysis files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .ANA. No wildcard characters are allowed in the file specification.

By default, no data analysis file is created.

/CHECK
/NOCHECK
Controls whether all run-time checks are suppressed. The /NOCHECK qualifier is equivalent to having all possible SUPPRESS pragmas in the source code.

Explicit use of the /CHECK qualifier overrides any occurrences of the pragmas SUPPRESS and SUPPRESS_ALL in the source code, without the need to edit the source code.

By default, run-time checks are suppressed only in cases where a pragma SUPPRESS or SUPPRESS_ALL appears in the source code.

See the *VAX Ada Language Reference Manual* for more information on the pragmas SUPPRESS and SUPPRESS_ALL.

/COPY_SOURCE (D)
/NOCOPY_SOURCE
Controls whether a copied source file is created in the current program library when a compilation unit is compiled without error. The ACS RECOMPILE command requires that a copied source file exist in the current program library; the ACS COMPILE command uses the copied

source file if it cannot find an external source file when it is recompiling an obsolete unit or completing an incomplete generic instantiation (see Chapter 3). Copied source files may also be used by the VMS Debugger (see Chapter 6).

By default, a copied source file is created in the current program library when a unit is compiled without error.

/DEBUG[=(option[,...])] (D)
/NODEBUG
Controls which compiler debugging options are provided. You can debug VAX Ada programs with the VMS Debugger (see Chapters 6 and 7). You can request the following options:

ALL	Provides both SYMBOLS and TRACEBACK.
NONE	Provides neither SYMBOLS nor TRACEBACK.
[NO]SYMBOLS	Controls whether debugger symbol records are included in the object file.
[NO]TRACEBACK	Controls whether traceback information (a subset of the debugger symbol information) is included in the object file.

By default, both debugger symbol records and traceback information are included in the object file (/DEBUG=ALL, or equivalently: /DEBUG).

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)
Controls whether a diagnostics file containing compiler messages and diagnostic information is created. The diagnostics file is supported only for use with Digital layered products, such as the VAX Language-Sensitive Editor.

One diagnostics file is created for each source file that is compiled. The default directory for diagnostics files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .DIA. No wildcard characters are allowed in the file specification.

By default, no diagnostics file is created.

/ERROR_LIMIT[=n] (D)
/NOERROR_LIMIT
Controls whether execution of the ADA command for a given compilation unit is terminated upon the occurrence of the nth E-level error within that unit.

Error counts are not accumulated across a sequence of compilation units. If the /ERROR_LIMIT=n option is specified, each compilation unit may have up to n – 1 errors without terminating the compilation. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues.

The /ERROR_LIMIT=0 option is equivalent to ERROR_LIMIT=1.

By default, execution of the ADA command is terminated for a given compilation unit upon the occurrence of the 30th E-level error within that unit (equivalent to /ERROR_LIMIT=30).

/LIST[=file-spec]
/NOLIST (D)
Controls whether a listing file is created. One listing file is created for each source file compiled. The default directory for listing files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the ADA command does not create a listing file.

/LOAD[= option]
/NOLOAD
Controls whether the current program library is updated with the successfully processed units contained in the specified source files. Depending on other qualifiers specified (or not specified) with the ADA command, processing can involve full compilation, syntax checking only, and so on. The /NOLOAD qualifier causes the units in the specified source files to be processed, but prevents the current program library from being updated. For example, this effect allows you to obtain a machine code listing for a unit that has already been compiled into the program library without affecting the library.

You can specify the following option:

[NO]REPLACE	Controls whether a unit added to the current program library replaces an existing unit with the same name. If you specify the NOREPLACE option, the unit will be added to the current program library only if no existing unit has the same name, except if the new unit is the missing body of an existing specification, or vice versa.

By default, the current program library is updated with the successfully processed units, and a unit added to the current program library will replace an existing unit with the same name (/LOAD=REPLACE).

/MACHINE_CODE
/NOMACHINE_CODE (D)
Controls whether generated machine code (approximating assembler notation) is included in the listing file.

By default, generated machine code is not included in the listing file.

/NOTE_SOURCE (D)
/NONOTE_SOURCE
Controls whether the file specification of the source file is noted in the program library when a unit is compiled without error. The ACS COMPILE command uses this information to locate revised source files.

By default, the file specification of the source file is noted in the program library when a unit is compiled without error.

/OPTIMIZE[=(option[,...])]
/NOOPTIMIZE
Controls the level of optimization that is applied in producing the compiled code. You can specify one of the following primary options:

TIME	Provides full optimization with time as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(SPACE) in the source code.
SPACE	Provides full optimization with space as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(TIME) in the source code.

DEVELOPMENT — Suggested when active development of a program is in progress. Provides some optimization, but development considerations and ease of debugging take preference over optimization. This option overrides pragmas that establish a dependence on a subprogram or generic body (the pragmas INLINE and INLINE_GENERIC), and thus reduces the need for recompilations when such bodies are modified. This option also disables generic code sharing.

NONE — Provides no optimization. Suppresses inline expansions of subprograms and generics, including those specified by the pragmas INLINE and INLINE_GENERIC. Suppresses occurrences of the pragma SHARE_GENERIC and disables generic code sharing.

The /NOOPTIMIZE qualifier is equivalent to /OPTIMIZE=NONE.

By default, the ADA command applies full optimization with time as the primary optimization criterion (like /OPTIMIZE=TIME, but observing uses of the pragma OPTIMIZE).

The /OPTIMIZE qualifier also has a set of secondary options that you can use separately or together with the primary options to override the default behavior for inline expansion (generic and subprogram) and generic code sharing.

The INLINE secondary option can have the following values (see the *VAX Ada Run-Time Reference Manual* for more information about inline expansion):

NONE — Disables subprogram and generic inline expansion. This option overrides any occurrences of the pragmas INLINE or INLINE_GENERIC in the source code, without your having to edit the source file. It also disables implicit inline expansion of subprograms. (*Implicit inline expansion* means that the compiler assumes a pragma INLINE for certain subprograms as an optimization.) A call to a subprogram or an instance of a generic in another unit is not expanded inline, regardless of the /OPTIMIZE options in effect when that unit was compiled.

NORMAL	Provides normal subprogram and generic inline expansion. Subprograms to which an explicit pragma INLINE applies are expanded inline under certain conditions. In addition, some subprograms are implicitly expanded inline. The compiler assumes a pragma INLINE for calls to some small local subprograms (subprograms that are declared in the same unit as the unit in which the call occurs). Instances are compiled separately from the unit in which the instantiation occurred unless a pragma INLINE_GENERIC applies to the instance. If a pragma INLINE_GENERIC applies and the generic body has been compiled, the generic is expanded inline at the point of instantiation.
SUBPROGRAMS	Provides maximal subprogram inline expansion and normal generic inline expansion. In addition to the normal subprogram inline expansion that occurs when INLINE:NORMAL is specified, this option results in implicit inline expansion of some small subprograms declared in other units. The compiler assumes a pragma INLINE for any subprogram if it improves execution speed and reduces code size. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE were specified explicitly in the source code. With this option, generic inline expansion occurs in the same manner as for INLINE:NORMAL.
GENERICS	Provides normal subprogram inline expansion and maximal generic inline expansion. With this option, subprogram inline expansion occurs in the same manner as for INLINE:NORMAL. The compiler assumes a pragma INLINE_GENERIC for every instantiation in the unit being compiled unless an explicit pragma SHARE_GENERIC applies or a generic body is not available. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE_GENERIC were specified explicitly in the source code.

MAXIMAL	Provides maximal subprogram and generic inline expansion. Maximal subprogram inline expansion occurs as for INLINE:SUBPROGRAMS, and maximal generic inline expansion occurs as for INLINE:GENERICS.

The SHARE secondary option can have the following values:

NONE	Disables generic sharing. This option overrides the effect of any occurrences of the pragma SHARE_GENERIC in the source code, without your having to edit the source file. In addition, instances do not share code from previous instantiations.
NORMAL	Provides normal generic sharing. Normally, the compiler will not attempt to generate shareable code for an instance (code that can be shared by subsequent instantiations) unless an explicit pragma SHARE_GENERIC applies to that instance. However, an instance will attempt to share code that resulted from a previous instantiation to which the pragma SHARE_GENERIC applied.
MAXIMAL	Provides maximal generic sharing. The compiler assumes that a pragma SHARE_GENERIC applies to every instance in the unit being compiled unless an explicit pragma INLINE_GENERIC applies. Thus, an instance will attempt to share code that resulted from a previous instantiation or to generate code that can be shared by subsequent instantiations. SHARE:MAXIMAL cannot be used in combination with INLINE:GENERICS or INLINE:MAXIMAL.

By default, the /OPTIMIZE qualifier primary options have the following secondary-option values:

/OPTIMIZE=TIME	=(INLINE:NORMAL, SHARE:NORMAL)
/OPTIMIZE=SPACE	=(INLINE:NORMAL, SHARE:NORMAL)
/OPTIMIZE=DEVELOPMENT	=(INLINE:NONE, SHARE:NONE)
/OPTIMIZE=NONE	=(INLINE:NONE, SHARE:NONE)

See Chapter 3 for more information of the /OPTIMIZE qualifier and its options.

/SHOW[=option] (D)
/NOSHOW
Controls the listing file options included when a listing file is provided. You can specify one of the following options:

ALL	Provides all listing file options.
[NO]PORTABILITY	Controls whether a program portability summary is included in the listing file (see Chapter 5).
NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the ADA command provides a portability summary (/SHOW=PORTABILITY).

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)
Controls whether the source file is to be checked only for correct syntax. If you specify the /SYNTAX_ONLY qualifier, other compiler checks are not performed (for example, semantic analysis, type checking, and so on).

In the presence of the /LOAD=REPLACE qualifier (the default), the /SYNTAX_ONLY qualifier updates the current program library with syntax-checked-only units. The units are considered to be obsolete and must be subsequently recompiled.

In the presence of the /NOLOAD qualifier, the /SYNTAX_ONLY qualifier checks the syntax of the specified units but does not update the library.

By default, the compiler performs all compiler checks.

/WARNINGS[=(option[,...])]
/NOWARNINGS
Controls which categories of informational (I-level) and warning (W-level) messages are displayed and where those messages are displayed. You can specify any combination of the following message options:

WARNINGS: (*destination*[,...])
NOWARNINGS

WEAK_WARNINGS: (*destination*[,...])
NOWEAK_WARNINGS

SUPPLEMENTAL: (*destination*[,...])
NOSUPPLEMENTAL

COMPILATION_NOTES: (*destination*[,...])
NOCOMPILATION_NOTES

STATUS: (*destination*[,...])
NOSTATUS

The possible values of *destination* are ALL, NONE, or any combination of TERMINAL (terminal device), LISTING (listing file), and DIAGNOSTICS (diagnostics file). The message categories are summarized as follows (see Chapter 3 for more information):

WARNINGS	W-level: Indicates a definite problem in a legal program—for example, an unknown pragma.
WEAK_WARNINGS	I-level: Indicates a potential problem in a legal program—for example, a possible CONSTRAINT_ERROR at run time. These are the only kind of I-level messages that are counted in the summary statistics at the end of a compilation.
SUPPLEMENTAL	I-level: Additional information associated with preceding E-level or W-level diagnostics.
COMPILATION_NOTES	I-level: Information about how the compiler translated a program, such as record layout, parameter-passing mechanisms, or decisions made for the pragmas INLINE, INTERFACE, or the import-subprogram pragmas.
STATUS	I-level: End-of-compilation statistics and other messages.

The defaults are as follows:

```
/WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)
```

If you specify only some of the message categories with the /WARNINGS qualifier, the default values for other categories are used.

Examples

1. `$ ADA RESERVATIONS,RESERVATIONS__CANCEL`

   Compiles the compilation units contained in the two files RESERVATIONS.ADA and RESERVATIONS__CANCEL.ADA, in the order given.

2. `$ ADA/LIST/SHOW=ALL SCREEN_IO_,SCREEN_IO`

   Compiles the compilation units contained in the two files SCREEN_IO_.ADA and SCREEN_IO.ADA, in the order given. The /LIST qualifier creates the listing files SCREEN_IO_.LIS and SCREEN_IO.LIS in the current default directory. The /SHOW=ALL qualifier causes all listing file options to be provided in the listing files.

3. `$ ADA/OPT=INLINE:MAX/WARN=COMPILATION_NOTES SCREEN_IO_, SCREEN_IO`

   Compiles the compilation units contained in the files SCREEN_IO_.ADA and SCREEN_IO.ADA, in the order given. The /OPTIMIZE qualifier specifies maximal subprogram and generic inline expansion. The /WARN=COMPILATION_NOTES qualifier gives information about how the compiler translated the program, including the decisions made for inline expansions.

4. `$ ADA/NOLOAD/LIST/MACHINE_CODE HOTEL`

   Compiles the compilation units contained in the file HOTEL.ADA and generates a machine code listing, but does not update the current program library.

5. `$ ADA/WARNINGS=COMPILATION_NOTES/LIST STACKS, SUM`

   Compiles the compilation units contained in the files STACKS.ADA and SUM.ADA, giving information about record layout, parameter-passing mechanisms, inline expansions, and so on.

ATTACH

Enables you to switch control of your terminal from your current process running the program library manager to another process in your job. See also the ACS SPAWN command and the *VMS DCL Dictionary*.

Format

ATTACH *process-name*

Prompts

_Process:

Command Parameters

process-name
Specifies the name of the process to which the connection is to be made. Process names can contain from 1 to 15 alphanumeric characters. If a connection to the specified process cannot be made, an error message is displayed. You cannot connect to the process if any of the following conditions apply:

- The process is your current process.
- The process is not part of your current job.
- The process does not exist.

Description

The ACS ATTACH command allows you to connect your input stream to another process. You can use the ATTACH command to change control from one subprocess to another subprocess or to the parent process.

When you enter the ATTACH command, the parent or "source" process is put into a hibernation state, and your input stream is connected to the specified destination process. You can use the ATTACH command to connect to a subprocess that is part of a current job left hibernating as a result of an

ATTACH

ACS SPAWN or DCL SPAWN/WAIT command, or of another ACS or DCL ATTACH command, as long as the connection is valid. (No connection can be made to the current process, to a process that is not part of the current job, or to a process that does not exist.)

You can also use the ATTACH command in conjunction with the ACS SPAWN or DCL SPAWN/WAIT command to return to a parent process without terminating the created subprocess. See the description of the ACS SPAWN command for more details.

Example

```
ACS> ATTACH JONES_1
$
```

Switches control of the terminal to the process JONES_1.

CHECK

Forms the execution closure of one or more specified units and checks whether the set of units in the closure is complete and current. The ACS CHECK command searches the current program library (and all parent libraries, in the case of a sublibrary) for all units in the closure.

Format

CHECK *unit-name[,...]*

Command Qualifiers	Defaults
/[NO]LOG	/NOLOG
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units in the current program library whose closure is to be checked. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

CHECK

Description

The ACS CHECK command goes through the following steps:

1. Forms the execution closure of the specified units.
2. Determines whether each unit in the closure is in the program library and is current. Units entered from other program libraries, as well as those compiled or copied into the current program library, are checked.
3. Identifies any unit in the closure that is not in the program library.
4. Identifies any unit in the closure that is obsolete and must be recompiled.
5. If all of the units in the closure are in the program library and are current, issues an informational message.

Command Qualifiers

/LOG
/NOLOG (D)
Controls whether a list of all the units in the closure is displayed in addition to a message indicating the result of the CHECK command.

By default, only a message indicating the result of the CHECK command is displayed.

/OUTPUT=file-spec
Requests that the CHECK command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the CHECK command output is written to SYS$OUTPUT.

Examples

1.
```
ACS> CHECK SCREEN_IO
%I, All units current, no recompilations required
```

   Shows that all the units in the closure of SCREEN_IO are defined in the current program library and are current.

2.
```
ACS> CHECK RESERVATIONS
%E, Separate procedure body SCREEN_IO.OUTPUT not found in library
%E, Obsolete units are detected

%E, The following units need to be recompiled:
    RESERVATIONS
        package body                16-Apr-1989 13:34
    RESERVATIONS.RESERVE
        procedure body              16-Apr-1989 13:34
    RESERVATIONS.RESERVE.BILL
        procedure body              16-Apr-1989 13:35
    RESERVATIONS.CANCEL
        procedure body              16-Apr-1989 13:36
```

   Checks the closure of RESERVATIONS and finds that subunit SCREEN_IO.OUTPUT is missing from the current program library, and the body and subunits of RESERVATIONS are obsolete (they must be recompiled).

COMPILE

Forms the closure of one or more specified units. Compiles, from external source files, any unit in the closure (except entered units) that was revised since that unit was last compiled into the current program library. Recompiles, from external or copied source files, any unit in the closure that needs to be made current. Completes any incomplete generic instantiations.

Format

COMPILE *unit-name[,...]*

Command Qualifiers	Defaults
/AFTER=time	See text.
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/BATCH_LOG=file-spec	See text.
/[NO]CHECK	See text.
/CLOSURE	See text.
/COMMAND[=file-spec]	See text.
/[NO]CONFIRM	/NOCONFIRM
/[NO]COPY_SOURCE	/COPY_SOURCE
/[NO]DEBUG[=(option[,...])]	/DEBUG=ALL
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]ERROR_LIMIT[=n]	/ERROR_LIMIT=30
/[NO]KEEP	/KEEP
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG	/NOLOG
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/NAME=job-name	See text.
/[NO]NOTE_SOURCE	/NOTE_SOURCE
/[NO]NOTIFY	/NOTIFY
/[NO]OPTIMIZE[=(option[,...])]	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRELOAD	/NOPRELOAD
/[NO]PRINTER[=queue-name]	/NOPRINTER

/QUEUE=queue-name	/QUEUE=ADA$BATCH
/[NO]SHOW[=option]	/SHOW=PORTABILITY
/SPECIFICATION_ONLY	See text.
/SUBMIT	/SUBMIT
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/WAIT	See text.
/[NO]WARNINGS[=(option[,...])]	See text.
Positional Qualifiers	**Defaults**
/[NO]DATE_CHECK	/DATE_CHECK
/FORCE_BODY	See text.

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units in the current program library whose closure is to be processed with the ACS COMPILE command. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

Description

The ACS COMPILE command is useful for compiling and recompiling units as you revise the source files of an existing Ada program.

For each set of units specified, the COMPILE command goes through the following steps:

1. Forms the execution closure of the specified units.

2. Looks up the source file for each unit in the closure that has been compiled or copied (not entered) into the current program library. Unless otherwise specified with the SET SOURCE command, the source-file-directory search order is as follows:

   a. SYS$DISK:[ ] (the current default directory)

   b. ;0 (the directory that contained the file when it was last compiled), or node::;0 (if the file specification of the source file being compiled contains a node name)

   The search order takes precedence over the version number or creation date-time if different versions of a source file exist in two or more directories. Within any one directory, the version of a particular file that has the highest number is considered for compilation.

3. Compares the creation date-time of each source file with that of the version last noted in the program library by the /NOTE_SOURCE compiler qualifier (the qualifier is used with the DCL ADA and ACS COMPILE and RECOMPILE commands).

4. Processes external source files to account for new compilation units or unit dependences if the /PRELOAD qualifier is in effect.

5. Notes for compilation any source file whose creation date-time is later than that noted in the program library.

6. Identifies any obsolete or incomplete units in the closure.

   Note that if the program library manager cannot find external source files for recompilation, recompilation is done from copied source files. If a needed copied source file is missing, the file is identified and no recompilations or completions are done. Copied source files are created when the /COPY_SOURCE qualifier is in effect during compilation (the default for the DCL ADA and ACS LOAD and COMPILE commands).

   If the closure you are recompiling includes an obsolete entered unit, that unit is not affected by the COMPILE command; an error diagnostic is issued and the COMPILE command is not executed. You should recompile an obsolete entered unit in its own program library and then reenter it into the current program library before you try to recompile its dependent units in the current library.

7. Creates a DCL command file for the compiler. The file contains commands to compile the appropriate units from external source files and to recompile any obsolete units from external or copied source files, in the proper order. Entered units are not considered for compilation or recompilation. If you did not specify the /COMMAND qualifier, the command file is deleted after the COMPILE command is terminated, or the batch job finishes. If you did specify the /COMMAND qualifier, the command file is retained for future use, and the compiler is not invoked.
8. If you did not specify the /COMMAND qualifier, the VAX Ada compiler is invoked as follows:
   a. By default (COMPILE/SUBMIT), the compiler command file generated in step 7 is submitted as a batch job.
   b. If you specified the /WAIT qualifier, the command file is executed in a subprocess. You must wait for the compilation to terminate before entering another command. When you specify the COMPILE/WAIT command, process logical names are propagated to the subprocess generated to execute the command file.

Program library manager output originating before the compiler is invoked is reported to your terminal by default, or to a file specified with the /OUTPUT qualifier. Compiler diagnostics are reported to a log file by default, or to the terminal if the COMPILE command is executed in a subprocess (by way of the COMPILE/WAIT command).

See Chapter 3 for more information on the COMPILE command.

Command Qualifiers

/AFTER=time

Requests that the batch job be held until after a specific time when the COMPILE command is executed in batch mode (the default mode). If the specified time has already passed, or if the /AFTER qualifier is not specified, the job is queued for immediate processing.

You can specify either an absolute time or a combination of absolute and delta time. See the *VMS DCL Concepts Manual* (or type HELP Specify Date_Time at the DCL prompt) for complete information on specifying time values.

/ANALYSIS_DATA[=file-spec]
/NOANALYSIS_DATA (D)
Controls whether a data analysis file containing source code cross-reference and static analysis information is created. The data analysis file is supported only for use with Digital layered products, such as the VAX Source Code Analyzer.

One data analysis file is created for each source file that is compiled and for each unit that is recompiled. The default directory for data analysis files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .ANA. No wildcard characters are allowed in the file specification.

By default, no data analysis file is created.

/BATCH_LOG=file-spec
Provides a file specification for the batch log file when the COMPILE command is executed in batch mode (the default mode).

If you do not give a directory specification with the *file-spec* option, the batch log file is created by default in the current default directory. If you do not give a file specification with the *file-spec* option, the default file name is the job name specified with the /NAME=job-name qualifier. If no job name has been specified, the program library manager creates a file name comprising up to the first 39 characters of the first unit name specified. If no job name has been specified and there is a wildcard character in the first unit specified, the program library manager uses the default file name ACS_COMPILE. The default file type is .LOG. No wildcard characters are allowed in the file specification.

/CHECK
/NOCHECK
Controls whether all run-time checks are suppressed. The /NOCHECK qualifier is equivalent to having all possible SUPPRESS pragmas in the source code.

Explicit use of the /CHECK qualifier overrides any occurrences of the pragmas SUPPRESS and SUPPRESS_ALL in the source code, without the need to edit the source code.

By default, run-time checks are only suppressed in cases where a pragma SUPPRESS or SUPPRESS_ALL appears in the source code.

See the *VAX Ada Language Reference Manual* for more information on the pragmas SUPPRESS and SUPPRESS_ALL.

/CLOSURE
Causes the /SPECIFICATION_ONLY, /NODATE_CHECK, and /FORCE_BODY qualifiers to apply to all units in the closure of units named in the COMPILE command. (Without the /CLOSURE qualifier, these qualifiers apply only to the units named in the command.)

See the description of the /SPECIFICATION_ONLY qualifier in the list of command qualifiers; see the description of the /[NO]DATE_CHECK and /FORCE_BODY qualifiers in the list of positional qualifiers.

/COMMAND[=file-spec]
Controls whether the compiler is invoked as a result of the COMPILE command, and determines whether the command file generated to invoke the compiler is saved. If you specify the /COMMAND qualifier, the program library manager does not invoke the compiler, and the generated command file is saved for you to invoke or submit as a batch job.

The *file-spec* option allows you to enter a file specification for the generated command file. The default directory for the command file is the current default directory. By default, the program library manager provides a file name comprising up to the first 39 characters of the first unit name specified. If you use a wildcard character in the first unit name specified, the compiler uses the default name ACS_COMPILE. The default file type is .COM. No wildcard characters are allowed in the file specification.

By default, if you do not specify the *file-spec* option, the program library manager deletes the generated command file when the COMPILE command completes normally or is terminated.

/CONFIRM
/NOCONFIRM (D)
Controls whether the COMPILE command asks you for confirmation before performing a possibly lengthy operation. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/COPY_SOURCE (D)
/NOCOPY_SOURCE
Controls whether a copied source file is created in the current program library when a compilation unit is compiled without error. The ACS RECOMPILE command requires that a copied source file exist in the current program library; the ACS COMPILE command uses the copied source file if it cannot find an external source file when it is recompiling an obsolete unit or completing an incomplete generic instantiation (see Chapter 3). Copied source files may also be used by the VMS Debugger (see Chapter 6).

By default, a copied source file is created in the current program library when a unit is compiled without error.

/DEBUG[=(option[,...])] (D)
/NODEBUG
Controls which debugger compiler options are provided. You can debug VAX Ada programs with the VMS Debugger (see Chapters 6 and 7). You can request the following options:

ALL	Provides both SYMBOLS and TRACEBACK
NONE	Provides neither SYMBOLS nor TRACEBACK
[NO]SYMBOLS	Controls whether debugger symbol records are included in the object file
[NO]TRACEBACK	Controls whether traceback information (a subset of the debugger symbol information) is included in the object file

By default, both debugger symbol records and traceback information are included in the object files (/DEBUG=ALL, or equivalently: /DEBUG)

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)
Controls whether a diagnostics file containing compiler messages and diagnostic information is created. The diagnostics file is supported only

for use with Digital layered products, such as the VAX Language-Sensitive Editor.

One diagnostics file is created for each source file that is compiled and for each unit that is recompiled. The default directory for diagnostics files is the current default directory. The default file name is the name of the source file being compiled. The default file type of a diagnostics file is .DIA. No wildcard characters are allowed in the file specification.

By default, no diagnostics file is created.

/ERROR_LIMIT[=n] (D)
/NOERROR_LIMIT
Controls whether execution of the COMPILE command for a given compilation unit is terminated upon the occurrence of the nth E-level error within that unit.

Error counts are not accumulated across a sequence of compilation units. If the /ERROR_LIMIT=n option is specified, each compilation unit may have up to n − 1 errors without terminating the compilation. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues.

The /ERROR_LIMIT=0 option is equivalent to ERROR_LIMIT=1.

By default, execution of the COMPILE command is terminated for a given compilation unit upon the occurrence of the 30th E-level error within that unit (equivalent to /ERROR_LIMIT=30).

/KEEP (D)
/NOKEEP
Controls whether the batch log file generated is deleted after it is printed when the COMPILE command is executed in batch mode (the default mode).

By default, the log file is not deleted.

/LIST[=file-spec]
/NOLIST (D)
Controls whether a listing file is created. One listing file is created for each compilation unit (not file) compiled or recompiled by the COMPILE command.

The default directory for listing files is the current default directory. The default file name of a listing file corresponds to the name of its compilation unit and uses the VAX Ada file-name conventions described in Chapter 1. The default file type of a listing file is .LIS. No wildcard characters are allowed in the file specification.

By default, the COMPILE command does not create a listing file.

/LOG
/NOLOG (D)
Controls whether a list of all the units that must be compiled or recompiled is displayed.

By default, a list of the units that must be compiled or recompiled is not displayed.

/MACHINE_CODE
/NOMACHINE_CODE (D)
Controls whether generated machine code (approximating assembler notation) is included in the listing file.

By default, generated machine code is not included in the listing file.

/NAME=job-name
Specifies a string to be used as the job name and as the file name for the batch log file when the COMPILE command is executed in batch mode (the default mode). The job name can have from 1 to 39 characters.

By default, if you do not specify the /NAME qualifier, the program library manager creates a job name comprising up to the first 39 characters of the first unit name specified. If you do not specify the /NAME qualifier, but use a wildcard character in the first unit name specified, the compiler uses the default name ACS_COMPILE. In these cases, the job name is also the file name of the batch log file.

/NOTE_SOURCE (D)
/NONOTE_SOURCE
Controls whether the file specification of the source file is noted in the program library when a unit is compiled without error. The COMPILE command uses this information to locate revised source files.

By default, the file specification of the source file is noted in the current program library when a unit is compiled without error.

/NOTIFY (D)
/NONOTIFY
Controls whether a message is broadcast when the COMPILE command is executed in batch mode (the default mode). The message is broadcast to any terminal at which you are logged in, notifying you that your job has been completed or terminated.

By default, a message is broadcast.

/OPTIMIZE[=(option[,...])]
/NOOPTIMIZE
Controls the level of optimization that is applied in producing the compiled code. You can specify one of the following primary options:

TIME	Provides full optimization with time as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(SPACE) in the source code.
SPACE	Provides full optimization with space as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(TIME) in the source code.
DEVELOPMENT	Suggested when active development of a program is in progress. Provides some optimization, but development considerations and ease of debugging take preference over optimization. This option overrides pragmas that establish a dependence on a subprogram or generic body (the pragmas INLINE and INLINE_GENERIC), and thus reduces the need for recompilations when such bodies are modified. This option also disables generic code sharing.
NONE	Provides no optimization. Suppresses inline expansions of subprograms and generics, including those specified by the pragmas INLINE and INLINE_GENERIC. Suppresses occurrences of the pragma SHARE_GENERIC and disables generic code sharing.

The /NOOPTIMIZE qualifier is equivalent to /OPTIMIZE=NONE.

By default, the COMPILE command applies full optimization with time as the primary optimization criterion (like /OPTIMIZE=TIME, but observing uses of the pragma OPTIMIZE).

The /OPTIMIZE qualifier also has a set of secondary options that you can use separately or together with the primary options to override the default behavior for inline expansion (generic and subprogram) and generic code sharing.

The INLINE secondary option can have the following values (see the *VAX Ada Run-Time Reference Manual* for more information about inline expansion):

NONE	Disables subprogram and generic inline expansion. This option overrides any occurrences of the pragmas INLINE or INLINE_GENERIC in the source code, without your having to edit the source file. It also disables implicit inline expansion of subprograms. (*Implicit inline expansion* means that the compiler assumes a pragma INLINE for certain subprograms as an optimization.) A call to a subprogram or an instance of a generic in another unit is not expanded inline, regardless of the /OPTIMIZE options in effect when that unit was compiled.
NORMAL	Provides normal subprogram and generic inline expansion. Subprograms to which an explicit pragma INLINE applies are expanded inline under certain conditions. In addition, some subprograms are implicitly expanded inline. The compiler assumes a pragma INLINE for calls to some small local subprograms (subprograms that are declared in the same unit as the unit in which the call occurs). Instances are compiled separately from the unit in which the instantiation occurred unless a pragma INLINE_GENERIC applies to the instance. If a pragma INLINE_GENERIC applies and the generic body has been compiled, the generic is expanded inline at the point of instantiation.

SUBPROGRAMS

Provides maximal subprogram inline expansion and normal generic inline expansion.

In addition to the normal subprogram inline expansion that occurs when INLINE:NORMAL is specified, this option results in implicit inline expansion of some small subprograms declared in other units. The compiler assumes a pragma INLINE for any subprogram if it improves execution speed and reduces code size. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE were specified explicitly in the source code.

With this option, generic inline expansion occurs in the same manner as for INLINE:NORMAL.

GENERICS

Provides normal subprogram inline expansion and maximal generic inline expansion.

With this option, subprogram inline expansion occurs in the same manner as for INLINE:NORMAL.

The compiler assumes a pragma INLINE_GENERIC for every instantiation in the unit being compiled unless an explicit pragma SHARE_GENERIC applies or a generic body is not available. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE_GENERIC were specified explicitly in the source code.

MAXIMAL

Provides maximal subprogram and generic inline expansion.

Maximal subprogram inline expansion occurs as for INLINE:SUBPROGRAMS, and maximal generic inline expansion occurs as for INLINE:GENERICS.

The SHARE secondary option can have the following values:

NONE

Disables generic sharing. This option overrides the effect of any occurrences of the pragma SHARE_GENERIC in the source code, without your having to edit the source file. In addition, instances do not share code from previous instantiations.

NORMAL	Provides normal generic sharing. Normally, the compiler will not attempt to generate shareable code for an instance (code that can shared by subsequent instantiations) unless an explicit pragma SHARE_GENERIC applies to that instance. However, an instance will attempt to share code that resulted from a previous instantiation to which the pragma SHARE_GENERIC applied.
MAXIMAL	Provides maximal generic sharing. The compiler assumes that a pragma SHARE_GENERIC applies to every instance in the unit being compiled unless an explicit pragma INLINE_GENERIC applies. Thus, an instance will attempt to share code that resulted from a previous instantiation or to generate code that can be shared by subsequent instantiations. SHARE:MAXIMAL cannot be used in combination with INLINE:GENERICS or INLINE:MAXIMAL.

By default, the /OPTIMIZE qualifier primary options have the following secondary-option values:

/OPTIMIZE=TIME	=(INLINE:NORMAL, SHARE:NORMAL)
/OPTIMIZE=SPACE	=(INLINE:NORMAL, SHARE:NORMAL)
/OPTIMIZE=DEVELOPMENT	=(INLINE:NONE, SHARE:NONE)
/OPTIMIZE=NONE	=(INLINE:NONE, SHARE:NONE)

See Chapter 3 for more information about the /OPTIMIZE qualifier and its options.

/OUTPUT=file-spec

Requests that any program library manager output generated before the compiler is invoked be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the COMPILE command output is written to SYS$OUTPUT.

/PRELOAD
/NOPRELOAD (D)
Controls whether the COMPILE command processes external source files so that new compilation units or unit dependences introduced in those files—or any new source files previously processed by the ACS LOAD or DCL ADA command—are accounted for. Preload processing involves the partial compilation and syntax checking of the following files:

- Any external source file whose creation date-time is later than that noted in the program library
- Any new units introduced into the closure of units specified by way of modifications to the known external source files (preload processing does not include new external source files that are not already accounted for in the program library)

Preload processing is done immediately, after the creation date-time of each external source file is checked, and before the usual COMPILE compilations and recompilations are performed. If you have also specified the /CONFIRM qualifier, you are prompted for confirmation for each external file to be preloaded.

By default, the COMPILE command does not process external source files to account for new compilation units or unit dependences.

/PRINTER[=queue-name]
/NOPRINTER (D)
Controls whether the batch job log file is queued for printing when the COMPILE command is executed in batch mode (the default mode).

The /PRINTER qualifier allows you to specify a particular print queue. The default print queue for the log file is SYS$PRINT.

By default, the log file is not queued for printing. If you specify the /NOPRINTER qualifier, the /KEEP qualifier is assumed.

/QUEUE=queue-name
Specifies the batch job queue in which the job is entered when the COMPILE command is executed in batch mode (the default mode).

By default, if the /QUEUE qualifier is not specified, the program library manager first checks whether the logical name ADA$BATCH is defined. If it is, the program library manager enters the job in the queue specified. Otherwise the job is placed in the default system batch job queue, SYS$BATCH.

/SHOW[=option] (D)
/NOSHOW
Controls the listing file options included when a listing file is provided. You can specify one of the following options:

ALL	Provides all listing file options.
[NO]PORTABILITY	Controls whether a program portability summary is included in the listing file (see Chapter 5).
NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the COMPILE command provides a portability summary (/SHOW=PORTABILITY).

/SPECIFICATION_ONLY
Causes only the specifications of the units specified to be considered for compilation. You can use the /CLOSURE qualifier with the /SPECIFICATION_ONLY qualifier to force only the specifications in the execution closure of the specified units to be considered for compilation.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, all of the specifications, bodies, and subunits in the execution closure of the units specified are considered for compilation.

/SUBMIT
Directs the program library manager to submit the command file generated for the compiler to a batch queue. You can continue to enter commands in your current process without waiting for the batch job to complete. The compiler output is written to a log file.

By default, the program library manager submits the command file generated for the compiler to a batch queue.

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)
Controls whether the source file is to be checked only for correct syntax. If you specify the /SYNTAX_ONLY qualifier, other compiler checks are not performed (for example, semantic analysis, type checking, and so on), and the program library is not updated.

By default, the compiler performs all checks.

/WAIT
Directs the program library manager to execute the command file generated for the compiler in a subprocess. Execution of your current process is suspended until the subprocess completes. The compiler output is written directly to your terminal. Note that process logical names are propagated to the subprocess generated to execute the command file.

By default, the program library manager submits the command file generated for the compiler to a batch queue (by way of the COMPILE/SUBMIT command).

/WARNINGS[=(option[,...])]
/NOWARNINGS
Controls which categories of informational (I-level) and warning (W-level) messages are displayed and where those messages are displayed. You can specify any combination of the following message options:

WARNINGS: (*destination*[,...])
NOWARNINGS

WEAK_WARNINGS: (*destination*[,...])
NOWEAK_WARNINGS

SUPPLEMENTAL: (*destination*[,...])
NOSUPPLEMENTAL

COMPILATION_NOTES: (*destination*[,...])
NOCOMPILATION_NOTES

STATUS: (*destination*[,...])
NOSTATUS

The possible values of *destination* are ALL, NONE, or any combination of TERMINAL (terminal device), LISTING (listing file), and DIAGNOSTICS (diagnostics file). The message categories are summarized as follows (see Chapter 3 for more information):

WARNINGS	W-level: Indicates a definite problem in a legal program—for example, an unknown pragma.
WEAK_WARNINGS	I-level: Indicates a potential problem in a legal program—for example, a possible CONSTRAINT_ERROR at run time. These are the only kind of I-level messages that are counted in the summary statistics at the end of a compilation.
SUPPLEMENTAL	I-level: Additional information associated with preceding E-level or W-level diagnostics.
COMPILATION_NOTES	I-level: Information about how the compiler translated a program, such as record layout, parameter-passing mechanisms, or decisions made for the pragmas INLINE, INTERFACE, or the import-subprogram pragmas.
STATUS	I-level: End of compilation statistics and other messages.

The defaults are as follows:

```
/WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)
```

If you specify only some of the message categories with the /WARNINGS qualifier, the default values for the other categories are used.

Positional Qualifiers

/DATE_CHECK (D)
/NODATE_CHECK

Controls whether the COMPILE command checks the creation date and time of source files to determine whether any source files have been revised but not compiled into the current program library. If you specify the /NODATE_CHECK qualifier, the COMPILE command forces the compilation of every unit specified, even though the source file has not been revised since the unit was last compiled; bodies and subunits of the specified units are also recompiled as necessary, to make them current. Entered units are not

considered for compilation or recompilation when the /NODATE_CHECK qualifier is in effect.

If you specify the /NODATE_CHECK/CLOSURE qualifier, the COMPILE command forces the compilation of every unit in the execution closure of the units specified.

You can use the /NODATE_CHECK qualifier to force the compilation of a set of units using a particular combination of compiler qualifiers.

By default, the COMPILE command checks the creation date and time of source files (/DATE_CHECK), and compiles only the source files that were revised but not compiled into the current program library.

/FORCE_BODY

Forces the compilation or recompilation of the bodies of the specified compilation units, regardless of whether or not the external source files have been modified or the bodies are obsolete.

The /FORCE_BODY qualifier can have different effects depending on its position in the command line and its interaction with other qualifiers:

- If you append the /FORCE_BODY qualifier to the COMPILE command string (as opposed to appending it to an individual unit parameter), the COMPILE command forces the compilation of the bodies of each unit specified on the command line.
- If you append the /FORCE_BODY qualifier to an individual unit parameter, the COMPILE command forces the compilation of the body of only that unit.
- If you specify the /FORCE_BODY qualifier with the /CLOSURE qualifier, the COMPILE command forces the compilation of the bodies of all of the units in the execution closure of the units specified.

By default, if the /FORCE_BODY qualifier is omitted, the specifications, bodies, and subunits of all of the units in the execution closure of the units specified are considered for compilation or recompilation.

Examples

1.
```
ACS> COMPILE/LOG RESERVATIONS
%I, The following units will be compiled from source files
RESERVATIONS
    package specification          16-Apr-1989 12:37
        USER:[JONES.HOTEL]RESERVATIONS_.ADA
    package body                   16-Apr-1989 12:37
        USER:[JONES.HOTEL]RESERVATIONS.ADA

%I, The following units will be recompiled:
RESERVATIONS.RESERVE
    procedure body                 16-Apr-1989 12:37
RESERVATIONS.RESERVE.BILL
    procedure body                 16-Apr-1989 12:37
RESERVATIONS.RESERVE.CANCEL
    procedure body                 16-Apr-1989 12:37

%I, Job RESERVATIONS (queue ALL_BATCH, entry 218) started
    on FAST_BATCH
```

   Lists all units in the closure of unit RESERVATIONS that need to be compiled and recompiled, then submits the compiler command file generated by ACS as a batch job.

2.
```
ACS> COMPILE/NODATE_CHECK/CLOSURE/OPTIMIZE=DEVELOPMENT HOTEL
```

   Forces (/NODATE_CHECK) the compilation (from external source files) of all of the units in the execution closure (/CLOSURE) of the unit HOTEL with the /OPTIMIZE=DEVELOPMENT qualifier.

3.
```
ACS> COMPILE/PRELOAD QUEUE_MANAGER
```

   Determines the compilation order for the units in the execution closure of QUEUE_MANAGER, and compiles from external source files any units that have been modified.

CONVERT LIBRARY

Converts VAX Ada Version 1.n program libraries and sublibraries into VAX Ada Version 2.0 program libraries and sublibraries.

Format

CONVERT LIBRARY *directory-spec1 [directory-spec2]*

Command Qualifiers	Defaults
/PARENT_LIBRARY=directory-spec	See text.

Prompts

_Library:

Command Parameters

directory-spec1
Specifies a VAX Ada Version 1.n program library. The directory specification must contain a VMS directory name and, optionally, a device name (see the *VMS DCL Dictionary* for VMS directory naming conventions). The directory may be a subdirectory or a main (top-level) directory. No wildcard characters are allowed in the directory specification.

If this is the only parameter specified, then the files in the VAX Ada Version 1.n program library are converted in place (as opposed to being converted and copied to a second VMS directory).

directory-spec2
Specifies an empty VMS directory that will hold the converted files from the VAX Ada Version 1.n program library. The directory specification must contain a VMS directory name and, optionally, a device name (see the *VMS DCL Dictionary* for VMS directory naming conventions). The directory may be a subdirectory or a main (top-level) directory. No wildcard characters are allowed in the directory specification.

When you specify this parameter, the Version 1.n program library is converted to this directory, resulting in two libraries: an intact Version 1.n program library and a new Version 2.0 library.

Description

The format of VAX Ada Version 2.0 program libraries and sublibraries is different from the format of previous VAX Ada Version 1.n program libraries and sublibraries. The ACS CONVERT LIBRARY command converts a Version 1.n library to a Version 2.0 library.

NOTE

Before converting a library, make sure that the library is in a valid, consistent state by first entering an ACS VERIFY or ACS VERIFY/REPAIR command with the VAX Ada Version 1.n program library manager.

After the conversion (either in place or into another VMS directory), the Version 2.0 library contains the same units as the Version 1.n library. However, the Version 2.0 units are obsolete and must be recompiled using either the ACS COMPILE or RECOMPILE command.

If the conversion is interrupted, you can restart it. Restarting means that the conversion starts at the beginning, rather than starting where the interruption took place. If you are converting into another VMS directory, you must delete the contents of that directory or otherwise make sure that you specify an empty directory as the second parameter to the CONVERT LIBRARY command.

NOTE

If you are converting a library in place, you cannot revert back to the Version 1.n format once you have started converting the library to Version 2.0 format, and you cannot use the Version 2.0 format with a Version 1.n program library manager. If you expect that you will need to use Version 1.n of VAX Ada with a Version 1.n program library, be sure to back up the Version 1.n program library before converting it.

To convert a sublibrary, you must be sure that the parent library is a Version 2.0 library. If you are converting a tree of libraries and sublibraries, you should convert from the top down. To perform the necessary recompilations, the entire tree must have been converted.

When you convert a library that has entered units, note that the ACS CONVERT LIBRARY command handles entered units differently from nonentered units. You may need to manually enter or reenter units depending on how you convert the library into which the entered units are compiled and depending on how you make the entered units current.

For example, consider the following situation:

- The unit QUEUE_MANAGER is compiled into the Version 1.n program library DISK:[SHARE.ADALIB].
- The unit QUEUE_MANAGER is entered from the Version 1.n library DISK:[SHARE.ADALIB] into the Version 1.n library USER:[HOTEL.ADALIB].

The following series of commands requires no manual entering or reentering of QUEUE_MANAGER when the two libraries are converted:

```
ACS> CONVERT LIBRARY DISK:[SHARE.ADALIB]
ACS> SET LIBRARY DISK:[SHARE.ADALIB]
ACS> COMPILE *
ACS> CONVERT LIBRARY USER:[HOTEL.ADALIB] USER:[HOTEL.NEWLIB]
ACS> ACS SET LIBRARY USER:[HOTEL.NEWLIB]
ACS> ACS COMPILE *
```

When the library USER:[HOTEL.ADALIB] is converted, the library DISK:[SHARE.ADALIB] has already been converted to Version 2.0 format, and the unit QUEUE_MANAGER has been made current by the ACS COMPILE operation. Thus, during the conversion of USER:[HOTEL.ADALIB], the ACS CONVERT LIBRARY command enters the unit QUEUE_MANAGER into USER:[HOTEL.NEWLIB] in a current state, so that it is available to users of USER:[HOTEL.NEWLIB] without any more operations.

In the following series of commands, the unit QUEUE_MANAGER must be manually reentered:

```
ACS> CONVERT LIBRARY DISK:[SHARE.ADALIB] DISK:[SHARE.NEWLIB]
ACS> CONVERT LIBRARY USER:[HOTEL.ADALIB] USER:[HOTEL.NEWLIB]
ACS> SET LIBRARY DISK:[SHARE.NEWLIB]
ACS> COMPILE *
ACS> SET LIBRARY USER:[HOTEL.NEWLIB]
ACS> ENTER UNIT DISK:[SHARE.NEWLIB] QUEUE_MANAGER/REPLACE
ACS> COMPILE *
```

During the conversion of the library USER:[HOTEL.ADALIB], the unit QUEUE_MANAGER in library DISK:[SHARE.ADALIB] is still in a Version 1.n state. The ACS CONVERT LIBRARY command enters the unit QUEUE_MANAGER into the library USER:[HOTEL.NEWLIB] in an obsolete state. After QUEUE_MANAGER is made current in DISK:[SHARE.NEWLIB] with the ACS COMPILE operation, it is manually reentered into USER:[HOTEL.NEWLIB]. Note that because QUEUE_MANAGER already exists in the library USER:[HOTEL.NEWLIB], you must enter the ACS ENTER UNIT command with the /REPLACE qualifier. Alternatively, you can use the ACS REENTER command.

In the following series of comands, the unit QUEUE_MANAGER must be manually entered:

```
ACS> CONVERT LIBRARY DISK:[SHARE.ADALIB]
ACS> CONVERT LIBRARY USER:[HOTEL.ADALIB] USER:[HOTEL.NEWLIB]
ACS> SET LIBRARY DISK:[SHARE.ADALIB]
ACS> COMPILE *
ACS> SET LIBRARY USER:[HOTEL.NEWLIB]
ACS> ENTER UNIT DISK:[SHARE.ADALIB] QUEUE_MANAGER
ACS> ACS COMPILE *
```

During the conversion of the library USER:[HOTEL.ADALIB], the unit QUEUE_MANAGER in library DISK:[SHARE.ADALIB] is in an obsolete, Version 2.0 state. The ACS CONVERT LIBRARY command does not enter the unit QUEUE_MANAGER into the library USER:[HOTEL.NEWLIB]. After QUEUE_MANAGER is made current in DISK:[SHARE.ADALIB] with the ACS COMPILE operation, it must be manually entered into USER:[HOTEL.NEWLIB]. Because QUEUE_MANAGER did not previously exist in USER:[HOTEL.NEWLIB], you do not need to enter the ACS ENTER UNIT command with the /REPLACE qualifier. Also, you cannot use the ACS REENTER command in this situation.

Command Qualifiers

/PARENT_LIBRARY=directory-spec

Specifies the parent library if the library being converted is a sublibrary. The directory specification must contain a VMS directory name and, optionally, a device name (see the *VMS DCL Dictionary* for VMS directory naming conventions). The directory may be a subdirectory or a main (top-level) directory. No wildcard characters are allowed in the directory specification.

The parent library (the library specified by this qualifier) must be a Version 2.0 library.

Examples

1.
```
ACS> CONVERT LIBRARY USER:[JONES.V15LIB] USER:[JONES.V2LIB]
ACS> SET LIBRARY USER:[JONES.V2LIB]
ACS> COMPILE *
```

   Converts a VAX Ada Version 1.5 library to a VAX Ada Version 2.0 library. Because a separate directory specification is given for the Version 2.0 library, the Version 1.5 library remains intact.

   The ACS COMPILE operation makes the units in USER:[JONES.V2LIB] current. Note that entered units may need to be manually entered or reentered during a series of library conversion operations, depending on when the units are converted and made current in their original libraries.

2.
```
ACS> CONVERT LIBRARY DISK:[SMITH.ADALIB]
ACS> SET LIBRARY DISK:[SMITH.ADALIB]
ACS> RECOMPILE *
```

   This set of commands shows the conversion of a library in place. After the library is converted, its contents are made current by recompilation of everything in the library.

COPY FOREIGN

Copies a foreign (non-Ada) object file into the current program library. The file is used as a library body (body of a package, procedure, or function).

Format

COPY FOREIGN *file-spec unit-name*

Command Qualifiers	Defaults
/[NO]LOG	/NOLOG
/[NO]REPLACE	/NOREPLACE

Prompts

_File:
_Unit:

Command Parameters

file-spec
Specifies the object file containing the foreign body to be copied into the current program library. The default directory is the current default directory. The default file type is .OBJ. No wildcard characters are allowed in the file specification.

unit-name
Specifies the unit whose body is to be copied into the current program library with the ACS COPY FOREIGN command.

Description

The ACS COPY FOREIGN command copies a foreign (non-Ada) object file into the current program library. Because the file is used as a library body, the program library must contain a library specification for the unit, and the specification must contain the pragma INTERFACE and (if appropriate) a pragma IMPORT_FUNCTION, IMPORT_PROCEDURE, or IMPORT_VALUED_PROCEDURE for any procedure or function that the specification requires.

Once you supply a foreign body for a unit, the program library manager assumes that the body is current until you supply a new (Ada or foreign) definition of the body. Compiling the specification of the unit does not cause the body to become obsolete.

Command Qualifiers

/LOG
/NOLOG (D)
Controls whether the unit name and object-file name are displayed after the object file is copied.

By default, the unit name or object-file name is not displayed.

/REPLACE
/NOREPLACE (D)
Controls whether the specified file replaces a body that is already defined in the current program library for the unit name specified.

By default, the specified file does not replace a body that is already defined in the current program library for the unit name specified.

Example

```
ACS> COPY FOREIGN USER:[JONES.WORK]SQUARE SQR
```

Copies the object file SQUARE.OBJ from the directory USER:[JONES.WORK] into the current program library as the body of unit SQR. The specification of SQR must already be defined in the current program library.

COPY UNIT

Copies one or more units from another program library into the current program library.

Format

COPY UNIT *from-directory-spec unit-name[,...]*

Command Qualifiers	Defaults
/[NO]CLOSURE	/NOCLOSURE
/[NO]CONFIRM	/NOCONFIRM
/[NO]ENTERED[=library]	/ENTERED
/[NO]LOCAL	/LOCAL
/[NO]LOG	/NOLOG
/[NO]REPLACE	/NOREPLACE

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/SPECIFICATION_ONLY	See text.

Prompts

_Library:
_Unit:

Command Parameters

from-directory-spec
Specifies the program library or program sublibrary that contains the units to be copied into the current program library.

unit-name
Specifies one or more units to be copied into the current program library. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

Description

The ACS COPY UNIT command copies, into the current program library, each specified unit's specification and body (if any). If the specified unit is a subunit, the COPY UNIT command copies the subunit and any nested subunits. If you specify the /CLOSURE qualifier, the COPY UNIT command copies the closure of the set of units specified.

For each unit copied, the COPY UNIT command updates the current program library as follows:

1. Creates local copies of all associated files
2. Updates the library index file of the current program library to account for the new files, and notes the date and time the unit was last compiled into its original program library

Copying a unit that was entered into a program library produces a local copy of that unit.

The COPY UNIT command does not affect the program library from which a unit is copied. Modifying the unit in the original program library does not affect the copied unit.

Once a unit is copied to a given program library, it can be used as if it had been compiled locally.

Command Qualifiers

/CLOSURE
/NOCLOSURE (D)
Controls whether the COPY UNIT command copies the closure of the set of units specified into the current program library.

By default, only the specification and body of the units specified are copied.

/CONFIRM
/NOCONFIRM (D)
Controls whether the COPY UNIT command displays the name of each unit before copying, and requests you to confirm whether or not the unit should be copied. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the confirmation prompt is reissued.

By default, no confirmation is requested.

/ENTERED[=library] (D)
/NOENTERED
Controls whether entered units are copied. You can use the library option to copy units that were entered from a particular library. When you specify the /NOENTERED qualifier, only units that have been compiled or copied into the current program library are copied. Note that when you specify the /ENTERED qualifier, local units are copied unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including entered units, are copied.

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are copied. Note that when you specify the /LOCAL qualifier, entered units are copied unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including local units, are copied.

/LOG
/NOLOG (D)
Controls whether the name of a unit is displayed after it has been copied.

By default, the names of copied units are not displayed.

/REPLACE
/NOREPLACE (D)
Controls whether the unit to be copied replaces a unit of the same name that is already defined in the current program library.

By default, the unit to be copied does not replace a unit of the same name that is already defined in the current program library.

Positional Qualifiers

/BODY_ONLY
Copies only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the COPY UNIT command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the COPY UNIT command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the specification, as well as the body, is copied.

/SPECIFICATION_ONLY
Copies only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the COPY UNIT command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the COPY UNIT command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the body, as well as the specification, is copied.

Examples

1. ACS> COPY UNIT [SMITH.WORK.ADALIB] STACKS,SUM

   Copies the units STACKS and SUM, located in the program library [SMITH.WORK.ADALIB], into the current program library.

2. ACS> COPY UNIT/CLOSURE DISK:[SMITH.SHARE.ADALIB] QUEUE_MANAGER

   Copies the closure of unit QUEUE_MANAGER from DISK:[SMITH.SHARE.ADALIB] into the current program library.

3. ACS> COPY UNIT DISK:[PROJ.ADALIB] STACKS*

   Copies the specification, body, and all of the subunits of the unit STACKS from the program library DISK:[PROJ.ADALIB] to the current program library.

CREATE LIBRARY

Creates a new VAX Ada program library. To create a program sublibrary, use the ACS CREATE SUBLIBRARY command.

Note that you cannot create a program library across DECnet if a corresponding VMS directory does not already exist.

Format

CREATE LIBRARY *directory-spec*

Command Qualifiers	Defaults
/[NO]LOG	/LOG
/LONG_FLOAT=option	/LONG_FLOAT=G_FLOAT
/MEMORY_SIZE=n	/MEMORY_SIZE=2147483647
/[NO]PREDEFINED	/PREDEFINED
/PROTECTION=(code)	See text.
/SYSTEM_NAME=system	/SYSTEM_NAME=VAX_VMS

Prompts

_Library:

Command Parameters

directory-spec

Specifies the program library to be created. The directory specification must contain a VMS directory name and, optionally, a device name (see the *VMS DCL Concepts Manual* for VMS directory naming conventions). The directory may be a subdirectory or a main (top-level) directory. No wildcard characters are allowed in the directory specification.

The program libraries you create will typically be subdirectories of your main (top-level) directory. To create a program library as a top-level directory, you must have the necessary privileges. To create a subdirectory, you must have write access to the lowest level directory that currently exists.

CREATE LIBRARY

The directory specified to be a program library may be an existing empty directory, to allow you to use special ACL (access control list) options for that directory. See the *VMS DCL Concepts Manual* and the *VMS Access Control List Editor Manual* for more information on directory protection and ACL options.

Description

The AC CREATE LIBRARY command creates and initializes a new program library by performing the following steps:

1. Creates the specified VMS directory, unless it already exists. If the directory already exists before the CREATE LIBRARY command is entered, the original directory protection attributes are maintained. If the directory does not exist when the command is entered, the command creates the specified directory with default protection attributes (see the description of the /PROTECTION qualifier).
2. Creates a library index file (ADALIB.ALB) and a library version control file (ADA$LIB.DAT) in the program library.
3. Initializes the program library to the following system characteristics:

   ```
   LONG_FLOAT = G_FLOAT
   MEMORY_SIZE = 2147483647
   SYSTEM_NAME = VAX_VMS
   ```

   You change these characteristics with the ACS SET PRAGMA command or with the /SYSTEM_NAME qualifier that applies to the ACS CREATE LIBRARY, CREATE SUBLIBRARY, EXPORT, and LINK commands.
4. If the /PREDEFINED qualifier is specified (the default), enters into the newly created program library the VAX Ada predefined units (such as SYSTEM and TEXT_IO) that are located in the ADA$PREDEFINED program library on your system. This is equivalent to entering an ACS ENTER UNIT command for those predefined units. You can use the /NOPREDEFINED qualifier to change this default.

The CREATE LIBRARY command does not define a new program library to be the current program library. You must use the ACS SET LIBRARY command to define the current program library.

Command Qualifiers

/LOG (D)
/NOLOG
Controls whether the program library directory specification is displayed after the library has been created.

By default, the program library directory specification is displayed.

/LONG_FLOAT=option
Initializes the program library to a particular value of LONG_FLOAT. The possible values are D_FLOAT and G_FLOAT. The effect of this qualifier is equivalent to compiling a pragma LONG_FLOAT.

By default, if the /LONG_FLOAT qualifier is not specified, the program sublibrary is initialized to the value G_FLOAT.

/MEMORY_SIZE=n
Initializes the memory size of the program library to the value n. The effect of this qualifier is equivalent to compiling a pragma MEMORY_SIZE.

By default, if the /MEMORY_SIZE qualifier is not specified, the initial memory size of the program library is 2,147,483,647 bytes.

/PREDEFINED (D)
/NOPREDEFINED
Controls whether the VAX Ada predefined units located in the program library denoted by the logical name ADA$PREDEFINED are entered into the specified program library.

By default, the VAX Ada predefined units are entered into the specified program library.

/PROTECTION=(code)
Defines the file protection to be applied to the program library. File protection is specified as follows:

```
/PROTECTION=(SYSTEM:rwed,OWNER:rwed,GROUP:rwed,WORLD:rwed)
```

Refer to the *VMS DCL Concepts Manual* for complete information on the form and meaning of file protection codes.

If you want to deny all access to a category, you must specify the category name without a colon. For example:

```
/PROTECTION=(OWNER:RWE,GROUP,WORLD)
```

If you do not specify a value for each access category, or if you omit the /PROTECTION qualifier when you create the program library, standard VMS directory and file protection defaults are applied as follows:

- The directory protection defaults from the next-higher-level directory, less any delete access.
- Protection for the library index file (ADALIB.ALB) and library version control file (ADA$LIB.DAT) defaults from the process default protection (see the DCL SET PROTECTION/DEFAULT command).

See Chapter 5 for more information on program library protection.

/SYSTEM_NAME=system
Determines the target operating system for the program library. The possible system values are VAX_VMS and VAXELN.

By default, if the /SYSTEM_NAME qualifier is not specified, the initial target operating system is VAX_VMS.

Examples

1.
```
ACS> CREATE LIBRARY [JONES.HOTEL.ADALIB]
%I, Library USER:[JONES.HOTEL.ADALIB] created
```

   Creates the program library [JONES.HOTEL.ADALIB] on the default device, USER:.

2.
```
ACS> CREATE LIBRARY/PROTECTION=(S:RWE,O:RWED,G:RW,W) -
_ACS> [PROJ.ADALIB]
%I, Program library USER:[PROJ.ADALIB] created
```

   Creates the program library [PROJ.ADALIB] on the default device, USER. The /PROTECTION qualifier assigns the specified program library protection. This protection is applied to the library index file, the library version control file, and the directory file for the newly created program library.

CREATE SUBLIBRARY

Creates a new VAX Ada program sublibrary and establishes its parent program library.

Note that you cannot create a program sublibrary across DECnet if the corresponding VMS directory does not already exist.

Format

CREATE SUBLIBRARY *directory-spec*

Command Qualifiers	Defaults
/[NO]LOG	/LOG
/LONG_FLOAT=option	/LONG_FLOAT=G_FLOAT
/MEMORY_SIZE=n	/MEMORY_SIZE=2147483647
/PARENT=directory-spec	/PARENT=current-program-library
/PROTECTION=(code)	See text.
/SYSTEM_NAME=system	/SYSTEM_NAME=VAX_VMS

Prompts

_Sublibrary:

Command Parameters

directory-spec

Specifies the program sublibrary to be created. The directory specification must contain a VMS directory name and, optionally, a device name (see the *VMS DCL Concepts Manual* for VMS directory naming conventions). No wildcard characters are allowed in the directory specification.

You may use any valid VMS directory specification when creating a program sublibrary; however, the program sublibraries you create will typically be subdirectories of your main (top-level) directory.

The specified program sublibrary directory may be, but need not be, a subdirectory of the parent library directory.

CREATE SUBLIBRARY

The directory specified to be a program sublibrary may be an existing empty directory. This allows you to use special ACL (access control list) options for that directory. In that case, the CREATE SUBLIBRARY command makes the directory a program library. See the *VMS DCL Concepts Manual* and the *VMS Access Control List Editor Manual* for more information on directory protection and ACL options.

Description

The ACS CREATE SUBLIBRARY command creates and initializes a new program sublibrary by performing the following steps:

1. Checks that the parent library exists and is write accessible.
2. Creates the specified VMS directory, unless it already exists. If the directory already existed before the CREATE SUBLIBRARY command was entered, the original directory protection attributes are maintained. If the directory did not exist before the command was entered, the command creates the specified directory with default protection attributes (see the description of the /PROTECTION qualifier).
3. Creates a library index file (ADALIB.ALB) and a library version control file (ADA$LIB.DAT) in the program sublibrary.
4. Initializes the library index file to reference the parent program library as specified with the /PARENT qualifier. If the /PARENT qualifier is not used, the parent program library is the current program library.
5. Initializes the program sublibrary to the parent library's current values for LONG_FLOAT, MEMORY_SIZE, and SYSTEM_NAME.

Program sublibraries may be nested several levels deep. However, you should limit nesting to three or four levels for best performance. Note that the VMS operating system imposes limits on how deeply directories and subdirectories can be nested. This limit has an effect only if you use increasingly subordinate subdirectories for each sublibrary in your sublibrary tree.

The CREATE SUBLIBRARY command does not affect the definition of your current program library. If you want to define the newly created program sublibrary to be the current program library, you must use the ACS SET LIBRARY command.

Command Qualifiers

/LOG (D)
/NOLOG
Controls whether the program sublibrary directory specification is displayed after the sublibrary has been created.

By default, the program sublibrary directory specification is displayed.

/LONG_FLOAT=option
Initializes the program library to a particular value of LONG_FLOAT. The possible values are D_FLOAT and G_FLOAT.

By default, if the /LONG_FLOAT qualifier is not specified, the program sublibrary is initialized to the parent library's current value of LONG_ FLOAT.

/MEMORY_SIZE=n
Initializes the memory size of the created program sublibrary.

By default, if the /MEMORY_SIZE qualifier is not specified, the initial memory size of the program sublibrary is the parent library's current value of MEMORY_SIZE.

/PARENT=directory-spec
Specifies the program library or program sublibrary that is the immediate parent of the program sublibrary to be created.

By default, if the /PARENT qualifier is not specified, the parent is the current program library as established by the last ACS SET LIBRARY command.

/PROTECTION=(code)
Defines the file protection to be applied to the program sublibrary. File protection is specified as follows:

```
/PROTECTION=(SYSTEM:rwed,OWNER:rwed,GROUP:rwed,WORLD:rwed)
```

Refer to the *VMS DCL Concepts Manual* for complete information on the form and meaning of file protection codes.

If you want to deny all access to a category, you must specify the category name without a colon. For example:

```
/PROTECTION=(OWNER:RWE,GROUP,WORLD)
```

If you do not specify a value for each access category, or if you omit the /PROTECTION qualifier when you create the program library, standard VMS directory and file protection defaults are applied as follows:

- The directory protection defaults from the next-higher-level directory, less any delete access.
- Protection for the library index file (ADALIB.ALB) and library version control file (ADA$LIB.DAT) defaults from the process default protection (see the DCL SET PROTECTION/DEFAULT command in the *VMS DCL Dictionary*).

See Chapter 5 for more information on program library protection.

/SYSTEM_NAME=system
Initializes the target operating system of the program sublibrary. The possible system values are VAX_VMS and VAXELN.

By default, if the /SYSTEM_NAME qualifier is not specified, the initial target operating system is the parent library's current value of SYSTEM_NAME.

Examples

1.
```
ACS> CREATE SUBLIBRARY [JONES.TEMP.SUBLIB]
%I, Sublibrary USER:[JONES.TEMP.SUBLIB] created
```

   Creates the program sublibrary [JONES.TEMP.SUBLIB] on the current default device. The parent library is the current program library.

2.
```
ACS> CREATE SUBLIBRARY/PARENT=[HOTEL.ADALIB] [JONES.LISTS.SUBLIB]
%I, Sublibrary USER:[JONES.LISTS.SUBLIB] created
```

   Creates the program sublibrary [JONES.LISTS.SUBLIB] on the current default device. The command defines [HOTEL.ADALIB] to be the parent library.

DELETE LIBRARY

Deletes a VAX Ada program library and all its units. To delete a program sublibrary, you must use the ACS DELETE SUBLIBRARY command.

NOTE

A program library does not contain any references to program sublibraries. When you enter the ACS DELETE LIBRARY command, you are not warned of the possible existence of any program sublibraries.

Format

DELETE LIBRARY *directory-spec*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/[NO]LOG	/LOG

Prompts

_Library:

Command Parameters

directory-spec
Specifies the program library directory to be deleted. The directory must be a VAX Ada program library; that is, it must have been created with the ACS CREATE LIBRARY command.

DELETE LIBRARY

Description

The ACS DELETE LIBRARY command performs the following steps:

1. Checks whether the directory specified to be deleted is a VAX Ada program library (has a valid library index file, ADALIB.ALB). If not, a message is issued and there is no further action.
2. If the specified directory is a VAX Ada program library, deletes the files needed for program library operations. For example, the library index file (ADALIB.ALB), library version control file (ADA$LIB.DAT), and all object (.OBJ), compilation unit (.ACU), and copied source (.ADC) files are deleted.
3. If the program library is empty after step 2 and has the appropriate protection, deletes the directory. If the directory is not empty, it is preserved and a message is issued. To delete the files and directory in that case, you must exit from the program library manager and use the DCL DELETE command.

   Note that, when a program library is created, the directory inherits the protection of its parent directory less any delete access by default. Before attempting to delete a program library that is delete protected against the owner, you must change the directory protection of the library with the DCL SET PROTECTION command. See Chapter 5 for more information on program library protection.

The DELETE LIBRARY command does not delete any program sublibraries of the specified program library.

You cannot use the DELETE LIBRARY command to delete a sublibrary.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)
Controls whether the DELETE LIBRARY command displays the name of the program library before deleting it and requests you to confirm whether or not the program library should be deleted. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/LOG (D)
/NOLOG
Controls whether the program library directory specification is displayed after the library has been deleted.

By default, the program library directory specification is displayed.

Example

```
ACS> DELETE LIBRARY/CONFIRM [JONES.TEMP.ADALIB]
USER:[JONES.TEMP.ADALIB], delete library? [N]: Y
%I, Library USER:[JONES.SCRATCH.ADALIB] deleted
```

Requests confirmation to delete the program library [JONES.TEMP.ADALIB]. After confirmation, the command deletes the library index file, library version control file, and all object, compilation unit, and copied source files. Because no other files remain in the program library directory, and the directory protection allows delete access, the directory is deleted and the name of the deleted program library is displayed.

DELETE SUBLIBRARY

Deletes a VAX Ada program sublibrary and all its units. To delete a program library, you must use the ACS DELETE LIBRARY command.

NOTE

A program sublibrary does not contain any references to nested program sublibraries. When you enter the ACS DELETE SUBLIBRARY command, you are not warned of the possible existence of any nested program sublibraries.

Format

DELETE SUBLIBRARY *directory-spec*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/[NO]LOG	/LOG

Prompts

_Sublibrary:

Command Parameters

directory-spec
Specifies the program sublibrary directory to be deleted. The directory must be a VAX Ada program sublibrary; that is, it must have been created with the ACS CREATE SUBLIBRARY command.

Description

The ACS DELETE SUBLIBRARY command performs the following steps:

1. Checks whether the directory specified to be deleted is a VAX Ada program sublibrary. If not, a message is issued and there is no further action.
2. If the specified directory is a VAX Ada program sublibrary, deletes the files needed for sublibrary operations. For example, the library index file (ADALIB.ALB), library version control file (ADA$LIB.DAT), and all object (.OBJ), compilation unit (.ACU), and copied source (.ADC) files are deleted.
3. If the program sublibrary is empty after step 2 and has the appropriate protection, deletes the directory. If the directory is not empty, it is preserved and a message is issued. To delete the files and directory in that case, you must exit from the program library manager and use the DCL DELETE command.

   Note that, when a program sublibrary is created, the directory inherits the protection of its parent directory less any delete access by default (note that the parent directory may not necessarily be the sublibrary's parent library). Before attempting to delete a program sublibrary that is delete protected against the owner, you must change the directory protection of the sublibrary with the DCL SET PROTECTION command. See Chapter 5 for more information on sublibrary protection.

The DELETE SUBLIBRARY command does not delete any nested program sublibraries of the specified program sublibrary.

The DELETE SUBLIBRARY command does not delete a program library.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)

Controls whether the DELETE SUBLIBRARY command displays the name of the program sublibrary before deleting it and requests you to confirm whether or not the program sublibrary should be deleted. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/LOG (D)
/NOLOG
Controls whether the program sublibrary directory specification is displayed after the sublibrary has been deleted.

By default, the program sublibrary directory specification is displayed.

Example

```
ACS> DELETE SUBLIBRARY/CONFIRM [JONES.LISTS.SUBLIB]
USER:[JONES.LISTS.SUBLIB], delete sublibrary? [N]: Y
%I, Sublibrary USER:[JONES.LISTS.SUBLIB] deleted
```

Requests confirmation to delete the sublibrary [JONES.LISTS.SUBLIB]. After confirmation, the command deletes the library index file and all object, compilation unit, and copied source files in [JONES.LISTS.SUBLIB]; it then deletes the program sublibrary.

DELETE UNIT

Deletes one or more units from the current program library, including references to units entered from another program library.

Format

DELETE UNIT *unit-name[,...]*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/[NO]ENTERED[=library]	/ENTERED
/[NO]LOCAL	/LOCAL
/[NO]LOG	/NOLOG

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/SPECIFICATION_ONLY	See text.

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units to be deleted from the current program library. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

DELETE UNIT

Description

The ACS DELETE UNIT command deletes, from the current program library, the specified unit's specification and body (if any). If you specify a subunit name, the DELETE UNIT command deletes the subunit and any nested subunits. The DELETE UNIT command deletes units that have been compiled, copied, or entered into the current program library.

NOTE

An ACS DELETE UNIT SYSTEM command deletes any unit called SYSTEM, be it predefined or user defined. Deleting the predefined unit SYSTEM can have major effects, such as not allowing you to use the predefined package TEXT_IO. If you accidentally delete the predefined package SYSTEM, you can restore it by entering the ACS ENTER UNIT command, and specifying the library denoted by the logical name ADA$PREDEFINED (see the ACS ENTER UNIT command for more information on entering units).

For each unit specified, the DELETE UNIT command updates the current program library as follows:

- Deletes the associated index entries in the library index file
- Deletes any associated files from the current program library

The DELETE UNIT command does not affect any files or index entries in program libraries other than the current program library.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)

Controls whether the DELETE UNIT command displays the unit name of each unit before deleting it, and requests you to confirm whether or not the unit should be deleted. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.

- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/ENTERED[=library] (D)
/NOENTERED
Controls whether entered units are deleted. You can use the library option to delete units that were entered from a particular library. When you specify the /NOENTERED qualifier, only units that have been compiled or copied into the current program library are deleted. Note that when you specify the /ENTERED qualifier, local units are deleted unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including entered units, are deleted.

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are deleted. Note that when you specify the /LOCAL qualifier, entered units are deleted unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and ENTERED).

By default, all units specified, including local units, are deleted.

/LOG
/NOLOG (D)
Controls whether the name of a unit is displayed after it has been deleted.

By default, the names of deleted units are not displayed.

DELETE UNIT

Positional Qualifiers

/BODY_ONLY
Deletes only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the DELETE UNIT command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the DELETE UNIT command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the specification, as well as the body, is deleted.

/SPECIFICATION_ONLY
Deletes only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the DELETE UNIT command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the DELETE UNIT command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the body, as well as the specification, is deleted.

Examples

1.
```
ACS> DELETE UNIT/LOG SCREEN_IO
%I, Package specification SCREEN_IO deleted
%I, Package body SCREEN_IO deleted
```

Deletes from the current program library the specification and body of SCREEN_IO, and displays the names of the components deleted.

2.
```
ACS> DELETE/BODY_ONLY/CONFIRM RESERVATIONS
RESERVATIONS, delete? [N]:Y
```

Deletes the body of RESERVATIONS from the current program library.

3. `ACS> DELETE UNIT STACKS*`

   Deletes the specification, body, and all of the subunits of the unit STACKS.

DIRECTORY

Displays information about one or more units in the current program library.

Format

DIRECTORY *[unit-name[,...]]*

Command Qualifiers	Defaults
/BRIEF	See text.
/[NO]ENTERED[=library]	/ENTERED
/[NO]LOCAL	/LOCAL
/FULL	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/SPECIFICATION_ONLY	See text.

Prompts

None.

Command Parameters

[unit-name[,...]]
Specifies one or more units in the current program library for which information is to be shown. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

Description

If you specify a unit name, the ACS DIRECTORY command displays information about the unit's specification and body, if the latter exists. If you specify a subunit name, the DIRECTORY command displays information about the subunit.

If you do not specify a unit name, the DIRECTORY command displays information about all of the units in the current program library, including entered units.

Units are listed by name in alphabetical order. Subunit names are shown using selected component notation.

The output of the DIRECTORY command depends on whether you specify the /BRIEF, /FULL, or no formatting qualifier. If you do not specify a qualifier, the DIRECTORY command displays (for each unit specified) the unit name, the kind of unit (for example, procedure body, generic package declaration, and so on), and the compilation date and time.

Command Qualifiers

/BRIEF
Lists only the names of the units specified.

/ENTERED[=library] (D)
/NOENTERED
Controls whether entered units are displayed. You can use the library option to display units that were entered from a particular library. When you specify the /NOENTERED qualifier, only units that have been compiled or copied into the current program library are displayed. Note that when you specify the /ENTERED qualifier, local units are displayed unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units, including entered units, are displayed.

/FULL
Lists (for each unit specified) the unit name, kind, compilation date and time, and the file specifications of the associated files. The file specifications of entered units are shown preceded with an at character (@).

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are displayed. Note that when you specify the /LOCAL qualifier, entered units are displayed unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including local units, are displayed.

/OUTPUT=file-spec
Requests that the DIRECTORY command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the DIRECTORY command output is written to SYS$OUTPUT.

Positional Qualifiers

/BODY_ONLY
Displays only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the DIRECTORY command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the DIRECTORY command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the specification, as well as the body, is displayed.

/SPECIFICATION_ONLY
Displays only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the DIRECTORY command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the DIRECTORY command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the body, as well as the specification, is displayed.

Examples

1.
```
ACS> DIRECTORY/NOENTERED/BRIEF *
  .
  .
  .
HOTEL
RESERVATIONS
RESERVATIONS.CANCEL
RESERVATIONS.RESERVE
RESERVATIONS.RESERVE.BILL
  .
  .
  .
Total of 13 units.
```

   Lists the names of all units and subunits that have been compiled or copied into the current program library.

2.
```
ACS> DIRECTORY SCREEN_IO*

SCREEN_IO
    package specification          16-Apr-1989 13:04
    package body                   16-Apr-1989 13:03

SCREEN_IO.INPUT
    procedure body                 16-Apr-1989 12:43

SCREEN_IO.INPUT.BUFFER
    function body                  16-Apr-1989 12:43

SCREEN_IO.OUTPUT
    procedure body                 16-Apr-1989 12:43

Total of 5 units.
```

Displays the unit name, unit kind, and date-time of the last compilation for all units in the current program library whose names start with SCREEN_IO.

3.
```
ACS> DIRECTORY/FULL SCREEN_IO.INPUT*

SCREEN_IO.INPUT
    procedure body                    16-Apr-1989 13:06
        SCREEN_IO__INPUT.ACU;4
        SCREEN_IO__INPUT.OBJ;4
        SCREEN_IO__INPUT.ADC;3
      @ USER:[JONES.HOTEL]SCREEN_IO__INPUT.ADA;2

SCREEN_IO.INPUT.BUFFER
    function body                     16-Apr-1989 13:06
        SCREEN_IO__BUFFER.ACU;4
        SCREEN_IO__BUFFER.OBJ;4
        SCREEN_IO__BUFFER.ADC;3
      @ USER:[JONES.HOTEL]SCREEN_IO__BUFFER.ADA;2

Total of 2 units.
```

Displays the unit name, unit kind, associated file specifications, and date-time of the last compilation for all of the units in the current program library whose names start with SCREEN_IO.INPUT. The file specifications listed are those for the compilation unit (.ACU), object (.OBJ), copied source (.ADC), and source (.ADA) files.

ENTER FOREIGN

Enters a reference to an external file into the current program library. The file is entered as a foreign (non-Ada) library body (the body of a package, procedure, or function). The file may be an object file, object library, shareable image library, shareable image, or linker options file.

Format

ENTER FOREIGN *file-spec unit-name*

Command Qualifiers	Defaults
/LIBRARY	See text.
/[NO]LOG	/NOLOG
/OBJECT	See text.
/OPTIONS	See text.
/[NO]REPLACE	/NOREPLACE
/SHAREABLE	See text.

Prompts

_File:
_Unit:

Command Parameters

file-spec
Specifies the file containing the foreign body to be entered into the current program library. The file may be a VMS object file, object library, shareable image library, shareable image, or linker options file.

The default directory is the current default directory. The default file type is .OBJ, unless the /LIBRARY, /OPTIONS, or /SHAREABLE qualifier is used. No wildcard characters are allowed in the file specification.

If the file is an object file, you can optionally use the /OBJECT qualifier. The default file type is .OBJ.

If the file is an object library or shareable image library, you must use the /LIBRARY qualifier. The default file type is .OLB.

If the file is a linker options file, you must use the /OPTIONS qualifier. The default file type is .OPT.

If the file is a shareable image, you must use the /SHAREABLE qualifier. The default file type is .EXE.

unit-name
Specifies the unit whose body is to be referenced with the ENTER FOREIGN command.

Description

The ACS ENTER FOREIGN command enters a reference to an external file, which then serves as a foreign (non-Ada) library body for an Ada compilation unit.

The program library must contain a library specification for the unit, and the specification must contain the pragma INTERFACE and (if appropriate) a pragma IMPORT_FUNCTION, IMPORT_PROCEDURE, or IMPORT_VALUED_PROCEDURE.

Once you supply a foreign body for a unit, the program library manager assumes that the body is current until you supply a new (Ada or foreign) definition of the body. Compiling the specification of the unit does not cause the body to become obsolete.

Command Qualifiers

/LIBRARY
Indicates that the associated input file is a VMS object library or shareable image library. The default file type is .OLB.

By default, if you do not specify the /LIBRARY qualifier, the file is assumed to be an object file with a default file type of .OBJ.

/LOG
/NOLOG (D)
Controls whether the unit name and associated file name are displayed after a foreign body has been entered.

By default, the unit name and associated file name are not displayed.

/OBJECT
Indicates that the associated input file is an object file. The default file type is .OBJ.

The /OBJECT qualifier is the default, if you do not specify a /LIBRARY, /OPTIONS, or /SHAREABLE qualifier.

/OPTIONS
Indicates that the associated input file is a VMS linker options file. The default file type is .OPT.

By default, if you do not specify the /OPTIONS qualifier, the file is assumed to be an object file with a default file type of .OBJ.

/REPLACE
/NOREPLACE (D)
Controls whether the foreign body to be entered replaces a body that is already defined in the current program library for the unit name specified.

By default, the foreign body to be entered does not replace a body that is already defined in the current program library for the unit name specified.

/SHAREABLE
Indicates that the associated input file is a VMS shareable image. The default file type is .EXE.

By default, if you do not specify the /SHAREABLE qualifier, the file is assumed to be an object file with a default file type of .OBJ.

Example

```
ACS> ENTER FOREIGN DISK:[SMITH.MATH]SQUARE SQR
```

Enters the object file SQUARE.OBJ from DISK:[SMITH.MATH] into the current program library, as the body of unit SQR. The specification of SQR is (as required) already defined in the program library.

ENTER UNIT

Enters references in the current program library to one or more units located in another program library.

Format

ENTER UNIT *from-directory-spec unit-name [,...]*

Command Qualifiers	Defaults
/[NO]CLOSURE	/NOCLOSURE
/[NO]CONFIRM	/NOCONFIRM
/[NO]ENTERED[=library]	/ENTERED
/[NO]LOCAL	/LOCAL
/[NO]LOG	/NOLOG
/[NO]REPLACE	/NOREPLACE

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/SPECIFICATION_ONLY	See text.

Prompts

_Library:
_Unit:

Command Parameters

from-directory-spec
Specifies the program library that contains the units to be referenced.

unit-name[,...]
Specifies one or more units to be entered into the current program library. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more detailed information on wildcard characters.)

Description

The ACS ENTER UNIT command enters into the current progam library each specified unit's specification and body (if any). If the specified unit is a subunit, the ENTER UNIT command enters the subunit and any nested subunits. If the /CLOSURE qualifier is specified, the ENTER UNIT command enters the closure of the set of units specified.

For each unit entered, the ENTER UNIT command updates the library index file of the current program library to refer to the unit and its associated files, and to include the date and time the unit was last compiled into its original program library.

An entered unit can be used as if had been compiled locally. In other words, a unit entered into a program library can be named in a **with** clause by a unit that has been compiled into that program library.

The ENTER UNIT command does not affect the program library from which a unit is entered. However, if an entered unit is subsequently compiled in its original program library, all references to that unit from other program libraries are invalidated. You must enter the ACS REENTER command to make the references current. (You may also need to then recompile any units that depend on the entered unit.)

The ACS COMPILE and RECOMPILE commands have no effect on entered units.

Command Qualifiers

/CLOSURE
/NOCLOSURE (D)
Controls whether the ENTER UNIT command enters the closure of the set of units specified into the current program library.

By default, only the specification and body of the units specified are entered.

/CONFIRM
/NOCONFIRM (D)
Controls whether the ENTER UNIT command displays the name of each unit before entering that unit into the current program library, and requests that you confirm whether or not that unit should be entered. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/ENTERED[=library] (D)
/NOENTERED
Controls whether entered units are entered. You can use the library option to enter units that were entered from a particular library. When you specify the /NOENTERED qualifier, only units that have been compiled or copied into the other program library are entered. Note that when you specify the /ENTERED qualifier, local units are entered unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units, including entered units, are entered.

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are entered. Note that when you specify the /LOCAL qualifier, entered units are entered unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including local units, are entered.

/LOG
/NOLOG (D)
Controls whether the name of a unit is displayed after it has been entered.

By default, the names of entered units are not displayed.

/REPLACE
/NOREPLACE (D)
Controls whether the unit to be entered replaces a unit of the same name that is already defined in the current program library.

By default, the unit to be entered does not replace a unit of the same name that is already defined in the current program library.

Positional Qualifiers

/BODY_ONLY
Enters only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the ENTER UNIT command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the ENTER UNIT command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the specification, as well as the body, is entered.

/SPECIFICATION_ONLY
Enters only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the ENTER UNIT command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the ENTER UNIT command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the body, as well as the specification, is entered.

Examples

1. ACS> ENTER UNIT [PROJ.MAIN_LIB] *

   Enters all of the units from the project library into the current program library.

2. ACS> ENTER UNIT/REPLACE DISK:[SMITH.SHARE.ADALIB] QUEUE_MANAGER

   Enters the unit QUEUE_MANAGER into the current program library from the library DISK:[SMITH.SHARE.ADALIB], replacing any previous index reference to that unit. If the /REPLACE qualifier had not been used, a previously existing reference to QUEUE_MANAGER would not have been replaced.

EXIT

Exits from the program library manager and returns control to DCL.

Format

EXIT

Prompts

None.

Command Parameters

None.

Description

The EXIT command allows you to exit from the program library manager when you are using it interactively. You can also use CTRL/Z for the same purpose.

Example

```
ACS> EXIT
$
```

Exits from the program library manager and returns control to DCL.

EXPORT

Creates an object file that contains the object code for all units in the closure of the list of units specified.

Format

EXPORT *unit-name[,...]*

Command Qualifiers	Defaults
/[NO]LOG	/NOLOG
/[NO]MAIN	/NOMAIN
/OBJECT=file-spec	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/SYSTEM_NAME=system	/SYSTEM_NAME=VAX_VMS

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units in the current program library whose closure will be used to create an object file.

If you specify the /MAIN qualifier:

- You can specify only one unit name.
- The generated object file contains the image transfer address, and thus can be used as a main program.
- The transfer address of the unit specified is used.

By default (or if you specify the /NOMAIN qualifier):

- You can specify more than one unit name. The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)
- The generated object file does not contain the image transfer address, and thus cannot be used as a main program. The exported units can be invoked by a non-Ada program.

Description

The ACS EXPORT command creates a concatenated object file for all the units in the closure of the list of units specified. The object file always contains code to elaborate any library packages that are exported.

Note that any exported units that will be called from a foreign module must contain the appropriate export pragma in the source code: EXPORT_FUNCTION, EXPORT_PROCEDURE, EXPORT_VALUED_PROCEDURE, EXPORT_OBJECT, PSECT_OBJECT, or EXPORT_EXCEPTION (see the *VAX Ada Language Reference Manual* and *VAX Ada Run-Time Reference Manual* for exact details).

Object files created by different invocations of the EXPORT command may include some code that is common—for example, if each closure includes the predefined unit TEXT_IO. In such cases, you will not be able to link those files into the same image. Whenever you think that closures may include units in common, you should specify all the units in a single EXPORT command line.

Command Qualifiers

/LOG
/NOLOG (D)
Controls whether a list of all the units included in the exported object file is displayed. The display shows the units according to the order of elaboration for the program.

By default, a list of the units included in the exported object file is not displayed.

/MAIN
/NOMAIN (D)
Controls whether the generated object file is to contain the image transfer address (of the first unit specified), and thus is to be a main program.

By default, the generated object file does not contain the image transfer address, and thus is not to be a main program.

/OBJECT=file-spec
Provides a file specification for the generated object file that is to be exported. The default directory is the current default directory. The default file type is .OBJ. No wildcard characters are allowed in the file specification.

By default, if you do not use the /OBJECT qualifier, a file name comprising up to the first 39 characters of the first unit name is provided.

/OUTPUT=file-spec
Requests that the EXPORT command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the EXPORT command output is written to SYS$OUTPUT.

/SYSTEM_NAME=system
Directs the program library manager to produce elaboration code for execution on a particular operating system; the possible system values are VAX_VMS and VAXELN. Note that when the value is VAXELN, the execution of elaboration code by non-Ada callers is not automatic; your program must take special action at run time to elaborate library packages. See the *VAXELN Ada User's Manual* for more information.

By default, if the /SYSTEM_NAME qualifier is not specified in the EXPORT command, the setting of the pragma SYSTEM_NAME for the current program library determines the target operating system environment.

Examples

1. ACS> EXPORT/MAIN HOTEL/OBJECT=EXP_HOTEL

   Creates the object file EXP_HOTEL.OBJ, which contains the code for all of the units in the execution closure of unit HOTEL, including any package elaboration code. Because the /MAIN qualifier is specified, the file created also contains the image transfer address.

2. ACS> EXPORT/SYSTEM_NAME=VAXELN HOTEL/OBJECT=VAXELN_HOTEL

   Creates the object file VAXELN_HOTEL, which contains the code for all of the units in the execution closure of unit HOTEL, including any package elaboration code. The elaboration code created is for a VAXELN target. See the *VAXELN Ada User's Manual* for information on how to prepare and run VAXELN Ada programs.

EXTRACT SOURCE

Creates a copy of the copied source files associated with the specified units. The specified units must be defined in the current program library.

Format

EXTRACT SOURCE *unit-name[,...]*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/[NO]ENTERED[=library]	/ENTERED
/[NO]LOCAL	/LOCAL
/[NO]LOG	/LOG

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/SPECIFICATION_ONLY	See text.

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units in the current program library whose copied source files are to be copied. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for detailed information on wildcard characters.)

Description

For each unit specified, the ACS EXTRACT SOURCE command creates a copy of the copied source files associated with the unit's specification and body. If a subunit name is specified, the EXTRACT SOURCE command creates a copy of the subunit's copied source file. The unit or subunit must be defined in the current program library.

The files are created in the current default directory. If they have less than or equal to 39 characters, the file names are the same as those of the corresponding copied source files in the current program library—that is, file names follow the file-name conventions defined in Chapter 1. If they have more than 39 characters, the program library manager generates a name. The file type of the created files is .ADA.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)

Controls whether the EXTRACT SOURCE command displays the name of each unit before creating a copy of the copied source files and requests that you confirm whether or not the unit should be copied. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/ENTERED[=library] (D)
/NOENTERED
Controls whether the source files for entered units are extracted. You can use the library option to extract units that were entered from a particular library. When you specify the /NOENTERED qualifier, only the source files for units that have been compiled or copied into the current program library are extracted. Note that when you specify the /ENTERED qualifier, local units are extracted unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, the source file for all units, including entered units, are extracted.

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are extracted. Note that when you specify the /LOCAL qualifier, the source files for entered units are extracted unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, the source files for all units specified, including local units, are extracted.

/LOG (D)
/NOLOG
Controls whether the names of units and extracted files are displayed after each unit's files are created.

By default, the names of units and files are displayed.

Positional Qualifiers

/BODY_ONLY
Extracts the source file for only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the EXTRACT SOURCE command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the EXTRACT SOURCE command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the source file for the specification, as well as the body, is extracted.

/SPECIFICATION_ONLY
Extracts the source file for only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the EXTRACT SOURCE command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the EXTRACT SOURCE command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the source file for the body, as well as the specification, is extracted.

Example

```
ACS> EXTRACT SOURCE TEXT_IO, STARLET
```

Creates in the current default directory the files TEXT_IO_.ADA, TEXT_IO.ADA, and STARLET_. These files are copies of the copied source files for, respectively, the specification and body of the predefined unit TEXT_IO and the specification of the predefined unit STARLET (TEXT_IO and STARLET were previously entered into the current program library).

HELP

Displays information on ACS commands and qualifiers.

Format

HELP *[keyword...]*

Prompts

None.

Command Parameters

[keyword...]
Specifies zero or more keywords that indicate what information you want. Information is located in a hierarchical manner, depending on the level of information required. The levels are as follows:

1. None—Lists the ACS commands and selected topics.
2. Topic—Provides information about the topic.
3. Command—Describes the command, its format, and parameters, and lists its qualifiers.
4. Command followed by a qualifier—Describes the use of the qualifier. For example, CHECK/LOG describes the use of the /LOG qualifier.

If you specify an asterisk (*) in place of any keyword, the HELP command displays all information available at that level.

You can specify percent signs (%) and asterisks (*) in the keywords as wildcard characters.

Example

```
ACS> HELP ENTER UNIT/CLOSURE
```

Displays information about the /CLOSURE qualifier to the ACS ENTER UNIT command.

LINK

Creates an executable image file for the specified units.

Format

LINK *unit-name [file-spec[,...]]*

LINK/NOMAIN *unit-name[,...] file-spec[,...]*

Command Qualifiers	Defaults
/AFTER=time	See text.
/BATCH_LOG=file-spec	See text.
/BRIEF	See text.
/COMMAND[=file-spec]	See text.
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=file-spec]	/NODEBUG
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE
/FULL	See text.
/[NO]KEEP	/KEEP
/[NO]LOG	/NOLOG
/[NO]MAIN	/MAIN
/[NO]MAP[=file-spec]	/NOMAP
/NAME=job-name	See text.
/[NO]NOTIFY	/NOTIFY
/OBJECT=file-spec	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	/QUEUE=SYS$BATCH
/SUBMIT	/See text.
/[NO]SYSLIB	/SYSLIB
/[NO]SYSSHR	/SYSSHR
/SYSTEM_NAME=system	/SYSTEM_NAME=VAX_VMS
/[NO]TRACEBACK	/TRACEBACK
/[NO]USERLIBRARY[=(table[,...])]	See text.
/WAIT	/WAIT

Parameter Qualifiers	Defaults
/INCLUDE=(object-file,...)	See text.
/LIBRARY	See text.
/OPTIONS	See text.
/SHAREABLE	See text.

Prompts

_Unit:
_File:

Command Parameters

unit-name[,...]
By default (or if you specify the /MAIN qualifier):

- You can specify only one unit, whose source code is written in Ada.
- The Ada main program, which must be a procedure or function with no parameters. If the main program is a function, it must return a value of a discrete type; the function value is used as the VMS image exit value.

If you specify the /NOMAIN qualifier:

- You can specify one or more units that are to be included in the executable image. The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)
- The image transfer address comes from one of the non-Ada files specified.

file-spec[,...]
Specifies a list of VMS object files, object libraries, shareable image libraries, shareable images, and linker option files that are to be used in linking the program. The default directory is the current default directory. The default file type is .OBJ, unless the /LIBRARY, /OPTIONS, or /SHAREABLE qualifier is used. No wildcard characters are allowed in a file specification.

If the file is an object library or shareable image library, you must use the /LIBRARY qualifier. The default file type is .OLB.

If the file is a linker options file, you must use the /OPTIONS qualifier. The default file type is .OPT.

If the file is a shareable image, you must use the /SHAREABLE qualifier. The default file type is .EXE.

If you specify the /NOMAIN qualifier, the image transfer address will come from one of the files (not units) specified.

Description

The ACS LINK command goes through the following steps:

1. If LINK/NOMAIN is not specified, checks that only one unit is specified and that it is an Ada main program.
2. Forms the execution closure of the main program (LINK/MAIN) or of the specified units (LINK/NOMAIN) and verifies that all units in the closure are present, current, and complete. If the program library manager detects an error, the operation is terminated before the VMS Linker is invoked.
3. Creates a DCL command file for the VMS Linker. The command file is deleted after the ACS LINK operation is completed or terminated, unless LINK/COMMAND is specified. If LINK/COMMAND is specified, the command file is retained for future use, and the linker is not invoked.
4. Creates an object file (to be linked with the program) that elaborates the library units in proper order at run time. If the /NOMAIN qualifier is not specified, the object file also contains the image transfer address. This object file is deleted after the ACS LINK operation is completed or terminated, unless the /COMMAND qualifier is specified. If the /COMMAND qualifier is specified, the object file is retained and the linker is not invoked.
5. Unless the /COMMAND qualifier is specified, invokes the VMS Linker as follows:
   a. By default (LINK/WAIT), the linker command file generated in step 3 is executed in a subprocess. You must wait for the link operation to terminate before entering another command. Note that when you specify the /WAIT qualifier (the default), process logical names are propagated to the subprocess generated to execute the command file.

b. If you specify the /SUBMIT qualifier, the linker command file is submitted as a batch job.

ACS output originating before the VMS Linker is invoked is reported to your terminal by default, or to a file specified with the /OUTPUT qualifier. Linker diagnostics are reported to your terminal, by default, or to a log file if the ACS LINK command is executed in batch mode (ACS LINK/SUBMIT).

See Chapter 4 for more information on the ACS LINK command. The VMS Linker is described in detail in the *VMS Linker Utility Manual*.

Command Qualifiers

/AFTER=time

Requests that the batch job be held until after a specific time, when the ACS LINK command is executed in batch mode (LINK/SUBMIT). If the specified time has already passed, the job is queued for immediate processing.

You can specify either an absolute time or a combination of absolute and delta time. See the *VMS DCL Concepts Manual* (or access the DCL HELP SPECIFY topic) for complete information on specifying time values.

/BATCH_LOG=file-spec

Provides a file specification for the batch log file when the ACS LINK command is executed in batch mode (LINK/SUBMIT).

If you do not give a directory specification with the *file-spec* option, the batch log file is created by default in the current default directory. If you do not give a file specification with the *file-spec* option, the default file name is the job name specified with the /NAME=job-name qualifier. If no job name has been specified, the program library manager creates a file name comprising up to the first 39 characters of the first unit name specified. If you specified LINK/NOMAIN and no job name and there is a wildcard character in the first unit specified, the program library manager uses the default file name ACS_LINK. The default file type is .LOG.

/BRIEF

Directs the linker to produce a brief image map file. The /BRIEF qualifier is valid only if you also specify the /MAP qualifier with the ACS LINK command. The /BRIEF qualifier is incompatible with the /FULL and /CROSS_REFERENCE qualifiers.

A brief image map file contains only the following sections:

- Object module synopsis
- Image synopsis
- Link run statistics

In contrast, the default image map file contains the preceding sections, as well as the program section synopsis and symbol definition section. See also the description of the /FULL qualifier.

/COMMAND[=file-spec]
Controls whether the linker is invoked as a result of the ACS LINK command, and determines whether the command file generated to invoke the linker is saved. If you specify the /COMMAND qualifier, the program library manager does not invoke the linker, and the generated command file is saved for you to invoke or submit as a batch job.

The *file-spec* option allows you to enter a file specification for the generated command file. The default directory for the command file is the current default directory. By default, the program library manager provides a file name comprising up to the first 39 characters of the first unit name specified. If you specified LINK/NOMAIN and you used a wildcard character in the first unit name specified, the compiler uses the default name ACS_ LINK. The default file type is .COM. No wildcard characters are allowed in the file specification.

By default, if the /COMMAND qualifier is not specified, the program library manager deletes the generated command file when the ACS LINK command completes normally or is terminated.

/CROSS_REFERENCE
/NOCROSS_REFERENCE (D)
Controls whether the image map file contains a symbol cross-reference. The /CROSS_REFERENCE qualifier is valid only if you also specify the /MAP qualifier in the ACS LINK command. The /CROSS_REFERENCE qualifier is incompatible with the /BRIEF qualifier.

When you specify the /CROSS_REFERENCE qualifier, the linker replaces the symbol definition section of the image map file with the symbol cross-reference section. The cross-reference section lists, in alphabetical order, the name of each global symbol, together with the following information about each:

- Its value
- The name of the first module in which it is defined
- The name of each module in which it is referenced

The number of symbols listed in the cross-reference section depends on whether the linker is generating a full image map or a default image map. In a full image map, this section includes global symbols from all modules in the image, including those extracted from all libraries. In a default image map, this section does not include global symbols from modules extracted from the default system libraries SYS$SHARE:IMAGELIB.OLB and SYS$SHARE:STARLET.OLB.

By default, the image map file does not contain a symbol cross-reference. In this case, if the linker is generating a default map or a full map, the map contains the symbol definition section instead of the symbol cross-reference section.

/DEBUG[=file-spec]
/NODEBUG (D)
Controls whether a debugger symbol table is included in the executable image file, and whether the VMS Debugger is invoked when the program is run.

The /DEBUG qualifier optionally allows you to specify an alternate debugger or dynamic performance analyzer. The default file type is .OBJ. See the *VMS Debugger Manual* for more information.

By default, no debugger symbol table is included in the executable image.

/EXECUTABLE[=file-spec] (D)
/NOEXECUTABLE
Controls whether the linker creates an executable image file and optionally provides a file specification for the file. The default file type is .EXE. No wildcard characters are allowed in the file specification.

You can use the /NOEXECUTABLE or /EXECUTABLE=NL: qualifier to test a set of qualifier options or input object modules without creating an image file. Using /EXECUTABLE=NL: is recommended, however, because the linker will not process certain qualifiers when the /NOEXECUTABLE qualifier is in effect.

By default, an executable image file is created with a file name comprising up to the first 39 characters of the first unit name specified.

/FULL
Directs the linker to produce a full image map file, which is the most complete image map. The /FULL qualifier is valid only if you also specify the /MAP qualifier with the ACS LINK command. Also, the /FULL qualifier is incompatible with the /BRIEF qualifier, but not with the /CROSS_ REFERENCE qualifier.

A full image map file contains the following sections:

- Object module synopsis
- Module relocatable reference synopsis
- Program section synopsis
- Symbol definitions
- Image section synopsis
- Symbols by value
- Module relocatable reference synopsis

In contrast, the default image map file does not contain the image section synopsis, the symbols by value, or the module relocatable reference synopsis sections.

Further, unlike the default image map, the full image map includes information about modules included from the system default libraries SYS$SHARE:STARLET.OLB and SYS$SHARE:IMAGELIB.OLB. Thus, the object module synopsis, program section synopsis, and symbols by name sections of a default image map do not contain information about modules included from these default libraries, whereas in a full image map they do.

/KEEP (D)
/NOKEEP
Controls whether the batch log file generated is deleted after it is printed when the ACS LINK command is executed in batch mode (LINK/SUBMIT).

By default, the log file is not deleted.

/LOG
/NOLOG (D)
Controls whether a list of all the units included in the executable image is displayed. The display shows the units according to the order of elaboration for the program.

By default, a list of all the units included in the executable image is not displayed.

/MAIN (D)
/NOMAIN
Controls where the image transfer address is to be found.

The /MAIN qualifier indicates that the VAX Ada unit specified determines the image transfer address and, hence, is to be a main program.

The /NOMAIN qualifier indicates that the image transfer address will come from one of the files specified, and not from one of the VAX Ada units specified.

By default (/MAIN), only one VAX Ada unit may be specified, and that unit must be a VAX Ada main program.

/MAP[=file-spec]
/NOMAP (D)
Controls whether the linker creates an image map file and optionally provides a file specification for the file. The default directory for the image map file is the current directory. The default file name comprises up to the first 39 characters of the first unit name specified. The default file type is .MAP. No wildcard characters are allowed in the file specification.

By default, no image map file is created.

/NAME=job-name
Specifies a string to be used as the job name and as the file name for the batch log file when the ACS LINK command is executed in batch mode (LINK/SUBMIT). The job name can have from 1 to 39 characters.

By default, if you do not specify the /NAME qualifier, the program library manager creates a job name comprising up to the first 39 characters of the first unit name specified. If you specify LINK/NOMAIN but do not specify the /NAME qualifier, and you use a wildcard character in the first unit name specified, the compiler uses the default name ACS_LINK. In these cases, the job name is also the file name of the batch log file.

/NOTIFY (D)
/NONOTIFY
Controls whether a message is broadcast when the ACS LINK command is executed in batch mode (LINK/SUBMIT). The message is broadcast to any terminal at which you are logged in, notifying you that your job has been completed or terminated.

By default, a message is broadcast.

/OBJECT=file-spec
Provides a file specification for the object file generated by the ACS LINK command. The file is retained by the program library manager only when the /COMMAND qualifier is used—that is, when the result of the ACS LINK operation is to produce a linker command file for future use, rather than to invoke the linker immediately.

The generated object file contains the code that directs the elaboration of library packages in the closure of the units specified. Unless you also specify the /NOMAIN qualifier, the object file also contains the image transfer address.

The default directory for the generated object file is the current default directory. The default file type is .OBJ. No wildcard characters are allowed in the file specification.

By default, if you do not specify the /OBJECT qualifier, the program library manager provides a file name comprising up to the first 39 characters of the first unit name specified.

By default, if you do not specify the /COMMAND qualifier, the program library manager deletes the generated object file when the ACS LINK command completes normally or is terminated.

/OUTPUT=file-spec
Requests that any ACS output generated before the linker is invoked be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the ACS LINK command output is written to SYS$OUTPUT.

/PRINTER[=queue-name]
/NOPRINTER (D)
Controls whether the log file is queued for printing when the ACS LINK command is executed in batch mode (LINK/SUBMIT) and the batch job is completed.

The /PRINTER qualifier allows you to specify a particular print queue. The default print queue for the log file is SYS$PRINT.

By default, the log file is not queued for printing. If you specify /NOPRINTER, /KEEP is assumed.

/QUEUE=queue-name
Specifies the batch job queue in which the job is entered when the ACS LINK command is executed in batch mode (LINK/SUBMIT).

By default, if the /QUEUE qualifier is not specified, the job is placed in the default system batch job queue, SYS$BATCH.

/SUBMIT
Directs the program library manager to submit the command file generated for the linker to a batch queue. You can continue to enter commands in your current process without waiting for the batch job to complete. The linker output is written to a batch log file.

By default, the generated command file is executed in a subprocess (LINK/WAIT).

/SYSLIB (D)
/NOSYSLIB
Controls whether the linker automatically searches the default system library for unresolved references. The default system library consists of the shareable image library SYS$LIBRARY:IMAGELIB.OLB and the object module library SYS$LIBRARY:STARLET.OLB.

By default, the default system library is automatically searched.

/SYSSHR (D)
/NOSYSSHR
Controls whether the linker automatically searches the default system shareable image library SYS$LIBRARY:IMAGELIB.OLB for unresolved references. If you specify the /NOSYSSHR qualifier, only SYS$LIBRARY:STARLET.OLB is searched for unresolved references.

By default, the default system shareable image library is searched.

/SYSTEM_NAME=system
Directs the program library manager to produce an image for execution on a particular operating system.

The possible system values are VAX_VMS and VAXELN. If VAX_VMS is specified, VMS versions of the Ada run-time library routines are used, and VMS-specific initialization code is generated. If VAXELN is specified, VAXELN versions of the Ada run-time library routines are used, and VAXELN-specific initialization code is generated. For more information on VAXELN Ada, see the *VAXELN Ada User's Manual.*

If the /SYSTEM_NAME qualifier is not specified in the ACS LINK command, the setting of the pragma SYSTEM_NAME for the current program library determines the target operating system environment.

/TRACEBACK (D)
/NOTRACEBACK
Controls whether the linker includes traceback information in the executable image file for run-time error reporting.

By default, traceback information is included in the executable image.

/USERLIBRARY[=(table[,...])]
/NOUSERLIBRARY
Controls whether the linker searches any user-defined default libraries after it has searched any specified user libraries. When you specify the /USERLIBRARY qualifier, the linker searches the process, group, and system logical name tables to find the file specifications of the user-defined libraries. (The discussion of the linker in the *VMS Linker Utility Manual* explains user-defined default libraries.) You can specify the following tables for the linker to search:

ALL	The linker searches the process, group, and system logical name tables for user-defined library definitions.
GROUP	The linker searches the group logical name table for user-defined library definitions.
NONE	The linker does not search any logical name table; this specification is equivalent to /NOUSERLIBRARY.

PROCESS	The linker searches the process logical name table for user-defined library definitions.
SYSTEM	The linker searches the system logical name table for user-defined library definitions.

By default, the linker assumes /USERLIBRARY=ALL.

/WAIT
Directs the program library manager to execute the command file generated for the linker in a subprocess. Execution of your current process is suspended until the subprocess completes. The linker output is written directly to your terminal. Note that process logical names are propagated to the subprocess generated to execute the command file.

By default, the program library manager executes the command file generated for the linker in a subprocess: you must wait for the subprocess to terminate before you can enter another command.

Parameter Qualifiers

/INCLUDE=(object-file,...)
Indicates that the associated input file is a VMS object module library or shareable image library with a default file type of .OLB, and that the named elements from that library should be linked with the main program named in the ACS LINK command.

/LIBRARY
Indicates that the associated input file is a VMS object module library or shareable image library to be searched for modules to resolve any undefined symbols in the input files. The default file type is .OLB.

By default, if you do not specify the /LIBRARY qualifier, the file is assumed to be an object file with a default file type of .OBJ.

/OPTIONS
Indicates that the associated input file is a VMS linker options file. The default file type is .OPT.

By default, if you do not specify the /OPTIONS qualifier, the file is assumed to be an object file with a default file type of .OBJ.

/SHAREABLE
Indicates that the associated input file is a VMS shareable image. The default file type is .EXE.

By default, if you do not specify the /SHAREABLE qualifier, the file is assumed to be an object file with a default file type of .OBJ.

Examples

1.
```
ACS> LINK HOTEL
```

   Forms the closure of the unit HOTEL, which is a VAX Ada main program, creates a linker command file and package elaboration file, then invokes the command file in a spawned subprocess.

2.
```
ACS> LINK/SUBMIT HOTEL NETWORK.OLB/LIBRARY,NET.OPT/OPTIONS
%I, Job HOTEL (queue ALL_BATCH, entry 134) started on FAST_BATCH
```

   Instructs the linker to link the closure of the VAX Ada main program HOTEL against the user library NETWORK.OLB, and to use the linker options file NET.OPT. The /SUBMIT qualifier causes the program library manager to submit the linker command file as a batch job.

3.
```
ACS> LINK/NOMAIN FLUID_VOLUME,COUNTER MONITOR.OBJ
```

   Links the VAX Ada units FLUID_VOLUME and COUNTER with the foreign object file MONITOR.OBJ. The /NOMAIN qualifier tells the linker that the image transfer address is in the foreign file.

4.
```
ACS> LINK HOTEL ELN$:RTL/INCLUDE=(KER$MSGDEF)
```

   Links the closure of the VAX Ada main program HOTEL against the message object file KER$MSGDEF from the VAXELN message library ELN$:RTL.OLB.

LOAD

Processes the Ada units contained in one or more source files. Processing involves determining the compilation order for the units in the files and invoking the VAX Ada compiler to partially compile the units. The partial compilation detects syntax errors and updates the current program library with unit dependence and source-file information.

Loaded units are considered to be obsolete and must be subsequently recompiled.

Format

LOAD *file-spec[,...]*

Command Qualifiers	**Defaults**
/AFTER=time	See text.
/BATCH_LOG=file-spec	See text.
/COMMAND[=file-spec]	See text.
/[NO]CONFIRM	/NOCONFIRM
/[NO]KEEP	/KEEP
/[NO]LOG	/NOLOG
/NAME=job-name	See text.
/[NO]NOTIFY	/NOTIFY
/OUTPUT=file-spec	See text.
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	See text.
/SUBMIT	/SUBMIT
/WAIT	See text.

Positional Qualifiers	**Defaults**
/BACKUP	See text.
/BEFORE[=time]	See text.
/BY_OWNER[=uic]	See text.
/[NO]COPY_SOURCE	/COPY_SOURCE
/CREATED	See text.
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS

LOAD

/[NO]ERROR_LIMIT[=n]	See text.
/EXCLUDE=(file-spec[,...])	See text.
/EXPIRED	See text.
/[NO]LIST[=file-spec]	/NOLIST
/MODIFIED	See text.
/[NO]NOTE_SOURCE	/NOTE_SOURCE
/[NO]REPLACE	/REPLACE
/SINCE	See text.
/[NO]WARNINGS[=(option[,...])]	See text.

Prompts

_File:

Command Parameters

file-spec[,...]
Specifies one or more VAX Ada source files to be loaded. If you do not specify a file type, the compiler uses the default file type of .ADA. Wildcard characters are allowed in the file specifications. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

Description

The ACS LOAD command invokes the VAX Ada compiler to partially compile the units contained in the specified files in any order. The partial compilation detects syntax errors and updates the current program library with unit dependence and source-file information. Units that are loaded into a program library are considered obsolete and must be subsequently recompiled. See Chapter 3 for more information on recompilation.

The LOAD command is useful for putting the units in a set of files into a program library for the first time.

The LOAD command does not check for missing or duplicate compilation units. (Units that have the same name are considered to be duplicates.) The LOAD command allows unit bodies to be loaded into the program library in the absence of their corresponding specifications. Similarly, subunits may be loaded into the library in the absence of their corresponding parent (or

ancestor) units. Because specifications, bodies, and subunits can be loaded in any order, the program library can be incomplete after a LOAD command has been executed. For example, the program library could contain a package body without a specification or a subunit without its corresponding parent unit.

For each set of files specified, the LOAD command goes through the following steps:

1. Resolves any wildcards in the list of source files specified. Within any one directory, the version of a particular file that has the highest number is considered for compilation.
2. Creates a DCL command file for the compiler. The file contains commands to compile the units in the source files. The command file is deleted after the LOAD command is terminated, unless you specified the /COMMAND qualifier. If you specified the /COMMAND qualifier, the command file is retained for future use, and the compiler is not invoked.
3. If you did not specify the /COMMAND qualifier, the VAX Ada compiler is invoked for syntax-only compilation as follows:
   a. By default (LOAD/SUBMIT), the compiler command file generated in step 2 is submitted as a batch job.
   b. If you specified the /WAIT qualifier, the command file is executed in a subprocess. You must wait for the compilation to terminate before entering another command. Note that when you specify the LOAD/WAIT command, process logical names are propagated to the subprocess generated to execute the command file.
   c. For each unit being compiled, the compiler checks to see if the unit is of the same name and kind as an existing unit in the current program library. If a unit has the same name and kind as an existing unit, a check is performed to see if the two units are identical; that is, to see if their source files have the same creation date and full file specification. If the two units are identical, the library is not updated with the new unit. If the two units are not identical or if the new unit is unique, the compiler updates the program library with the new unit.

Command Qualifiers

/AFTER=time
Requests that the batch job be held until after a specific time when the LOAD command is executed in batch mode (the default mode). If the specified time has already passed, or if the /AFTER qualifier is not specified, the job is queued for immediate processing.

You can specify either an absolute time or a combination of absolute and delta time. See the *VMS DCL Concepts Manual* (or type HELP Specify Date_Time at the DCL prompt) for complete information on specifying time values.

/BATCH_LOG=file-spec
Provides a file specification for the batch log file when the LOAD command is executed in batch mode (the default mode).

If you do not give a directory specification with the *file-spec* option, the batch log file is created by default in the current default directory. If you do not give a file specification with the *file-spec* option, the default file name is the job name specified with the /NAME=job-name qualifier. If no job name has been specified, the program library manager creates a file name comprising up to the first 39 characters of the first unit name specified. If no job name has been specified and there is a wildcard character in the first unit specified, the program library manager uses the default file name ACS_ LOAD The default file type is .LOG. No wildcard characters are allowed in the file specification.

/COMMAND[=file-spec]
Controls whether the LOAD operations are performed as a result of the LOAD command, and determines whether the command file generated to perform the LOAD operations is saved. If you specify the /COMMAND qualifier, the program library manager does not perform the LOAD operations, and the generated command file is saved for you to invoke or submit as a batch job.

The *file-spec* option allows you to enter a file specification for the generated command file. The default directory for the command file is the current default directory. By default, the program library manager provides a file name comprising up to the first 39 characters of the first unit name specified. If you use a wildcard character in the first unit name specified,

the compiler uses the default name ACS_LOAD. The default file type is .COM. No wildcard characters are allowed in the file specification.

By default, if you do not specify the *file-spec* option, the program library manager deletes the generated command file when the LOAD command completes normally or is terminated.

/CONFIRM
/NOCONFIRM (D)
Controls whether the LOAD command displays the name of each file before loading, and requests you to confirm whether or not the file should be processed. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/KEEP (D)
/NOKEEP
Controls whether the batch log file generated is deleted after it is printed when the LOAD command is executed in batch mode (the default mode).

By default, the log file is not deleted.

/LOG
/NOLOG
Controls whether a list of all the files that will be loaded is displayed.

By default, a list of the files that will be loaded is not displayed.

/NAME=job-name
Specifies a string to be used as the job name and as the file name for the batch log file when the LOAD command is executed in batch mode (the default mode). The job name can have from 1 to 39 characters.

By default, if you do not specify the /NAME qualifier, the program library manager creates a job name comprising up to the first 39 characters of the first file name specified. If you do not specify the /NAME qualifier, but use a wildcard character in the first file name specified, the compiler uses the default name ACS_LOAD. In these cases, the job name is also the file name of the batch log file.

/NOTIFY (D)
/NONOTIFY
Controls whether a message is broadcast when the NOTIFY command is executed in batch mode (the default mode). The message is broadcast to any terminal at which you are logged in, notifying you that your job has been completed or terminated.

By default, a message is broadcast.

/OUTPUT=file-spec
Requests that any program library manager output generated before the compiler is invoked be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the LOAD command output is written to SYS$OUTPUT.

/PRINTER[=queue-name]
/NOPRINTER (D)
Controls whether the batch job log file is queued for printing when the LOAD command is executed in batch mode (the default mode).

The /PRINTER qualifier allows you to specify a particular print queue. The default print queue for the log file is SYS$PRINT.

By default, the log file is not queued for printing. If you specify the /NOPRINTER qualifier, the /KEEP qualifier is assumed.

/QUEUE=queue-name
Specifies the batch job queue in which the job is entered when the LOAD command is executed in batch mode (the default mode).

By default, if the /QUEUE qualifier is not specified, the program library manager first checks whether the logical name ADA$BATCH is defined. If it is, the program library manager enters the job in the queue specified. Otherwise, the job is placed in the default system batch job queue, SYS$BATCH.

/SUBMIT
Directs the program library manager to submit the command file generated for the compiler to a batch queue. You can continue to enter commands in your current process without waiting for the batch job to complete. The compiler output is written to a log file.

By default, the program library manager submits the command file generated for the compiler to a batch queue.

/WAIT
Directs the program library manager to execute the command file generated for the compiler in a subprocess. Execution of your current process is suspended until the subprocess completes. The compiler output is written directly to your terminal. Note that process logical names are propagated to the subprocess generated to execute the command file.

By default, the program library manager submits the command file generated for the compiler to a batch queue (LOAD/SUBMIT).

Positional Qualifiers

/BACKUP
Selects files according to the dates of their most recent backups. Modifies the time value specified with the /BEFORE or /SINCE qualifier.

This qualifier is incompatible with the other qualifiers that also allow you to select files according to time attributes: /CREATED, /EXPIRED, and /MODIFIED. If you specify none of these four time qualifiers, the default is /CREATED.

/BEFORE[=time]
Selects only those files dated prior to the specified time. You can specify time as an absolute time, as a combination of absolute and delta times, or as one of the following keywords: TODAY (the default), TOMORROW, or YESTERDAY. See the *VMS DCL Concepts Manual* (or type HELP Specify Date_Time at the DCL prompt) for complete information on specifying time values.

You can specify one of the following qualifiers with the /BEFORE qualifier to indicate the time attribute to be used as the basis for selection: /BACKUP, /CREATED (the default), /EXPIRED, or /MODIFIED.

/BY_OWNER[=uic]
Selects only those files whose owner user identification code (UIC) matches the specified owner UIC. The default UIC is that of the current process.

/COPY_SOURCE (D)
/NOCOPY_SOURCE
Controls whether a copied source file is created in the current program library when a compilation unit is loaded without error. The ACS RECOMPILE command requires that a copied source file exist in the current program library; the ACS COMPILE command uses the copied source file if it cannot find an external source file when it is recompiling an obsolete unit or completing an incomplete generic instantiation (see Chapter 3). Copied source files may also be used by the VMS Debugger (see Chapter 6).

By default, a copied source file is created in the current program library when a unit is loaded without error.

/CREATED
Selects files based on their dates of creation. Modifies the time value specified with the /BEFORE or /SINCE qualifier.

This qualifier is incompatible with the other qualifiers that also allow you to select files according to time attributes: /BACKUP, /EXPIRED, and /MODIFIED. If you specify none of these four time qualifiers, the default is /CREATED.

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)
Controls whether a diagnostics file containing compiler messages and diagnostic information is created. The diagnostics file is supported only

for use with Digital layered products, such as the VAX Language-Sensitive Editor.

One diagnostics file is created for each source file that is compiled. The default directory for diagnostics files is the current default directory. The default file name is the name of the source file being compiled. The default file type of a diagnostics file is .DIA. No wildcard characters are allowed in the file specification.

By default, no diagnostics file is created.

/ERROR_LIMIT[=n]
/NOERROR_LIMIT
Controls whether execution of the LOAD command for a given compilation unit is terminated upon the occurrence of the nth E-level error within that unit.

Error counts are not accumulated across a sequence of compilation units. If the /ERROR_LIMIT=n option is specified, each compilation unit may have up to n − 1 errors without terminating the compilation. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues.

The /ERROR_LIMIT=0 option is equivalent to ERROR_LIMIT=1.

By default, execution of the COMPILE command is terminated for a given compilation unit upon the occurrence of the 30th E-level error within that unit (equivalent to /ERROR_LIMIT=30).

/EXCLUDE=(file-spec[,...])
Excludes the specified files from the LOAD operation. You can include a directory but not a device in the file specification. Wildcard characters are allowed in the file specification. However, you cannot use relative version numbers to exclude a specific version. If you provide only one file specification, you can omit the parentheses.

/EXPIRED
Selects files according to their expiration dates. (The expiration date is set with the DCL SET FILE/EXPIRATION_DATE command.) Modifies the time value specified with the /BEFORE or /SINCE qualifier.

This qualifier is incompatible with the other qualifiers that also allow you to select files according to time attributes: /BACKUP, /CREATED, and /MODIFIED. If you specify none of these four time qualifiers, the default is /CREATED.

/LIST[=file-spec]
/NOLIST (D)
Controls whether a listing file is created. One listing file is created for each compilation unit (not file) compiled by the LOAD command.

The default directory for listing files is the current default directory. The default file name of a listing file corresponds to the name of its compilation unit and uses the VAX Ada file-name conventions described in Chapter 1. The default file type of a listing file is .LIS. No wildcard characters are allowed in the file specification.

By default, the LOAD command does not create a listing file.

/MODIFIED
Selects files according to the dates on which they were last modified. Modifies the time value specified with the /BEFORE or /SINCE qualifier.

This qualifier is incompatible with the other qualifiers that also allow you to select files according to time attributes: /BACKUP, /CREATED, and /EXPIRED. If you specify none of these four time qualifiers, the default is /CREATED.

/NOTE_SOURCE (D)
/NONOTE_SOURCE
Controls whether the file specification of the source file is noted in the program library when a unit is loaded without error. The COMPILE command uses this information to locate revised source files.

By default, the file specification of the source file is noted in the current program library when a unit is compiled without error.

/REPLACE (D)
/NOREPLACE
Controls whether the loaded unit replaces a unit with the same name that is already defined in the current program library. If the /NOREPLACE qualifier is specified, and a unit already exists in the program library with the same name as a unit being loaded, a diagnostic message is issued, and the existing unit is not replaced.

By default, the loaded unit replaces a unit with the same name that is already defined in the current program library.

/SINCE
Selects only those files dated after the specified time. You can specify time as an absolute time, a combination of absolute and delta times, or as one of the following keywords: TODAY (the default), TOMORROW, or YESTERDAY. See the *VMS DCL Concepts Manual* (or type HELP Specify Date_Time at the DCL prompt) for complete information on specifying time values.

You can specify one of the following qualifiers with the /SINCE qualifier to indicate the time attribute to be used as the basis for selection: /BACKUP, /CREATED (the default), /EXPIRED, or /MODIFIED.

/WARNINGS[=(option[,...])]
/NOWARNINGS
Controls which categories of informational (I-level) and warning (W-level) messages are displayed and where those messages are displayed. You can specify any combination of the following message options:

WARNINGS: (*destination*[,...])
NOWARNINGS

WEAK_WARNINGS: (*destination*[,...])
NOWEAK_WARNINGS

SUPPLEMENTAL: (*destination*[,...])
NOSUPPLEMENTAL

COMPILATION_NOTES: (*destination*[,...])
NOCOMPILATION_NOTES

STATUS: (*destination*[,...])
NOSTATUS

The possible values of *destination* are ALL, NONE, or any combination of TERMINAL (terminal device), LISTING (listing file), and DIAGNOSTICS (diagnostics file). The message categories are summarized as follows (see Chapter 3 for more information):

WARNINGS	W-level: Indicates a definite problem in a legal program—for example, an unknown pragma.
WEAK_WARNINGS	I-level: Indicates a potential problem in a legal program—for example, a possible CONSTRAINT_ERROR at run time. These are the only kind of I-level messages that are counted in the summary statistics at the end of a compilation.
SUPPLEMENTAL	I-level: Additional information associated with preceding E-level or W-level diagnostics.
COMPILATION_NOTES	I-level: Information about how the compiler translated a program, such as record layout, parameter-passing mechanisms, or decisions made for the pragmas INLINE, INTERFACE, or the import-subprogram pragmas.
STATUS	I-level: End of compilation statistics and other messages.

The defaults are as follows:

```
/WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)
```

If you specify only some of the message categories with the /WARNINGS qualifier, the default values for the other categories are used.

Example

```
$ ACS LOAD/NOCOPY_SOURCE [JONES.NEW_UNITS]*
  .
  .
  .
$ ACS COMPILE/NOCOPY_SOURCE MAIN
  .
  .
  .
$ ACS LINK MAIN
```

This series of commands builds the program MAIN from a set of files that have never been previously compiled. The LOAD command puts syntax-checked, obsolete units into the current program library.

The COMPILE command recompiles the units from their original source files. The LINK command creates an executable image for the program MAIN. Note the use of /NOCOPY_SOURCE qualifiers to control the creation of copied source files.

MERGE

Moves one or more units from the current program sublibrary to its immediate parent program library.

Format

MERGE *unit-name[,...]*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/[NO]ENTERED[=library]	/ENTERED
/[NO]KEEP	/NOKEEP
/[NO]LOCAL	/LOCAL
/[NO]LOG	/NOLOG

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/SPECIFICATION_ONLY	See text.

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units, in the current program sublibrary, that are to be merged into the parent program library. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

Description

The ACS MERGE command moves each specified unit's specification and body (if any) from the current sublibrary to the parent library. If a subunit name is specified, the MERGE command moves the subunit into the parent library.

For each unit merged, the MERGE command moves its associated files into the parent library and updates the parent library's index file.

If the parent program library already has a version of the unit to be merged, the unit to be merged must have a more recent external source file.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)
Controls whether the MERGE command displays the name of each unit before merging and requests you to confirm whether or not the unit should be merged. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/ENTERED[=library] (D)
/NOENTERED
Controls whether entered units are merged. You can use the library option to merge units that were entered from a particular library. When you specify the /NOENTERED qualifier, only the units that have been compiled

or copied into the current program library are merged. Note that when you specify the /ENTERED qualifier, local units are merged unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units, including entered units, are merged.

/KEEP
/NOKEEP (D)
Controls whether a copy of a unit being merged is retained in the current program sublibrary after the merge operation.

By default, the unit is deleted from the program sublibrary after the merge operation.

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are merged. Note that when you specify the /LOCAL qualifier, entered units are merged unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including local units, are merged.

/LOG
/NOLOG (D)
Controls whether the name of each unit is displayed after it has been merged.

By default, the names of merged units are not displayed.

Positional Qualifiers

/BODY_ONLY
Merges only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the MERGE command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the MERGE command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the specification, as well as the body, is merged.

/SPECIFICATION_ONLY
Merges only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the MERGE command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the MERGE command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the body, as well as the specification, is merged.

Example

```
ACS> SET LIBRARY [JONES.HOTEL.SUBLIB]
%I, Current program library is USER:[JONES.HOTEL.SUBLIB])
ACS> SHOW LIBRARY/FULL

Program library USER:[JONES.HOTEL.SUBLIB]

    Sublibrary
        of USER:[HOTEL.ADALIB]
 .
 .
 .
ACS> MERGE RESERVATIONS.CANCEL
```

Establishes the sublibrary [JONES.HOTEL.SUBLIB] as the current program sublibrary. The SHOW LIBRARY/FULL command identifies the parent library [HOTEL.ADALIB]. The MERGE command copies the subunit RESERVATIONS.CANCEL from the current program sublibrary into the parent library, replacing any previous version of RESERVATIONS.CANCEL in the parent library, then deletes the original unit from the current program sublibrary.

If the copy of the unit in the parent library is newer than the unit in the sublibrary, the unit is not merged.

RECOMPILE

Enters an ACS CHECK command for the specified units, then recompiles (makes current) any obsolete unit that is part of the closure of the set of units specified. Obsolete entered units must be made current before you can use the ACS RECOMPILE command (see the Description section).

NOTE

To be recompiled, units must have previously been compiled with the /COPY_SOURCE qualifier (this is the default value of this qualifier).

Format

RECOMPILE *[unit-name[,...]]*

Command Qualifiers	Defaults
/AFTER=time	See text.
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/BATCH_LOG=file-spec	See text.
/[NO]CHECK	See text.
/CLOSURE	See text.
/COMMAND[=file-spec]	See text.
/[NO]CONFIRM	/NOCONFIRM
/[NO]COPY_SOURCE	/COPY_SOURCE
/[NO]DEBUG[=(option[,...])]	/DEBUG=ALL
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]ERROR_LIMIT[=n]	/ERROR_LIMIT=30
/[NO]KEEP	/KEEP
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG	/NOLOG
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/NAME=job-name	See text.
/[NO]NOTE_SOURCE	/NOTE_SOURCE
/[NO]NOTIFY	/NOTIFY
/[NO]OPTIMIZE[=(option[,...])]	See text.

/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRELOAD	/NOPRELOAD
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	/QUEUE=ADA$BATCH
/[NO]SHOW[=option]	/SHOW=PORTABILITY
/SPECIFICATION_ONLY	See text.
/SUBMIT	/SUBMIT
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/WAIT	See text.
/[NO]WARNINGS[=(option[,...])]	See text.
Positional Qualifiers	**Defaults**
/[NO]DATE_CHECK	/DATE_CHECK
/FORCE_BODY	See text.

Prompts

_Unit:

Command Parameters

[unit-name[,...]]

Specifies one or more units in the current program library whose closure is to be processed by the ACS RECOMPILE command. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for detailed information on wildcard characters.)

If you do not specify any units with the ACS RECOMPILE command, the command uses whatever units were involved with the most recent ACS CHECK command.

RECOMPILE

Description

The ACS RECOMPILE command is designed to be used when a unit or a set of units must be compiled again, but the original source code has not changed. Thus, the RECOMPILE command is useful for performing the following operations:

- To make an obsolete unit or set of units current (see Chapter 1 for definitions of obsolescence and currency).
- To complete incomplete generic instantiations, once the missing or changed generic body has been compiled into the current program library.
- To recompile units after the value of a global program library characteristic such as LONG_FLOAT or SYSTEM_NAME has been changed (for example, after you have used the ACS SET PRAGMA command).
- To obtain new versions of some units, compiled with a particular combination of compilation qualifiers (for example, /OPTIMIZE=SPACE, /CHECK, and so on). In this case, the units are not obsolete, but the RECOMPILE command, in combination with the /NODATE_CHECK qualifier, can be used to force the recompilation of the entire execution closure of a set of units.

For each set of units specified, the RECOMPILE command goes through the following steps:

1. Enter an ACS CHECK command:
   a. Forms the execution closure of the specified units.
   b. Determines whether each unit in the closure is in the program library and is current. Units entered from other program libraries as well as those compiled or copied into the current program library are checked.
   c. If all units in the closure are in the program library and are current, issues an informational message and terminates the operation.
   d. Identifies any unit in the closure that is missing from the current program library.
   e. Identifies any unit in the closure that is obsolete and must be recompiled.

2. If any units in the closure created by the CHECK command are obsolete, creates a DCL command file for the compiler. The file contains commands to compile the copied source file of each obsolete unit in the proper order. Entered units are not considered for recompilation. The command file is deleted after the RECOMPILE command is completed or terminated, unless the /COMMAND qualifier is specified. If the /COMMAND qualifier is specified, the command file is retained for future use, and the compiler is not invoked.
3. Unless the /COMMAND qualifier is specified, invokes the VAX Ada compiler as follows:
   a. By default (RECOMPILE/SUBMIT), the compiler command file generated in step 2 is submitted as a batch job.
   b. If you specify the /WAIT qualifier, the command file is executed in a subprocess. You must wait for the compilation to terminate before entering another command. When you specify the /WAIT qualifier, process logical names are propagated to the subprocess generated to execute the command file.

Note the use of copied source files in the recompilation. Files external to the current program library are ignored. If a copied source file needed for the recompilation is missing (because the /NOCOPY_SOURCE qualifier was specified in a previous compilation), the program library manager identifies the missing file, and the recompilation is not attempted. Thus, if you intend to use the RECOMPILE command, you should not compile units with the /NOCOPY_SOURCE qualifier.

If the closure you are recompiling includes an obsolete entered unit, that unit is not affected by the RECOMPILE command; an error diagnostic is issued and the RECOMPILE command is not executed. You should recompile an obsolete entered unit in its own program library and then reenter it into the current program library before you try to recompile its dependent units in the current library.

Program library manager output originating before the compiler is invoked is reported to your terminal by default, or to a file specified with the /OUTPUT qualifier. Compiler diagnostics are reported to a log file, by default, or to the terminal if the RECOMPILE command is executed in a subprocess (by way of the RECOMPILE/WAIT command).

See Chapter 3 for more information on the RECOMPILE command.

Command Qualifiers

/AFTER=time
Requests that the batch job be held until after a specific time when the RECOMPILE command is executed in batch mode (the default mode). If the specified time has already passed, or if the /AFTER qualifier is not specified, the job is queued for immediate processing.

You can specify either an absolute time or a combination of absolute and delta time. See the *VMS DCL Concepts Manual* (or type HELP Specify Date_Time at the DCL prompt) for complete information on specifying time values.

/ANALYSIS_DATA[=file-spec]
/NOANALYSIS_DATA (D)
Controls whether a data analysis file containing source code cross-reference and static analysis information is created. The data analysis file is supported only for use with Digital layered products, such as the VAX Source Code Analyzer.

One data analysis file is created for each copied source file that is recompiled. The default directory for data analysis files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .ANA. No wildcard characters are allowed in the file specification.

By default, no data analysis file is created.

/BATCH_LOG=file-spec
Provides a file specification for the batch log file when the RECOMPILE command is executed in batch mode (the default mode).

If you do not give a directory specification with the *file-spec* option, the batch log file is created by default in the current default directory. If you do not give a file specification with the *file-spec* option, the default file name is the job name specified with the /NAME=job-name qualifier. If no job name has been specified, the program library manager creates a file name comprising up to the first 39 characters of the first unit name specified. If no job name has been specified and there is a wildcard character in the first unit specified, the program library manager uses the default file name ACS_RECOMPILE. The default file type is .LOG. No wildcard characters are allowed in the file specification.

/CHECK
/NOCHECK
Controls whether all run-time checks are suppressed. The /NOCHECK qualifier is equivalent to having all possible SUPPRESS pragmas in the source code.

Explicit use of the /CHECK qualifier overrides any occurrences of the pragmas SUPPRESS and SUPPRESS_ALL in the source code, without the need to edit the source code.

By default, run-time checks are only suppressed in cases where a pragma SUPPRESS or SUPPRESS_ALL appears in the source code.

See the *VAX Ada Language Reference Manual* for more information on the pragmas SUPPRESS and SUPPRESS_ALL.

/CLOSURE
Causes the /SPECIFICATION_ONLY, /NODATE_CHECK, and /FORCE_BODY qualifiers to apply to all units in the closure of units named in the RECOMPILE command. (Without the /CLOSURE qualifier, these qualifiers apply only to the units named in the command.)

See the description of the /SPECIFICATION_ONLY qualifier in the list of command qualifiers; see the description of the /[NO]DATE_CHECK and /FORCE_BODY qualifiers in the list of positional qualifiers.

/COMMAND[=file-spec]
Controls whether the compiler is invoked as a result of the RECOMPILE command, and determines whether the command file generated to invoke the compiler is saved. If you specify the /COMMAND qualifier, the program library manager does not invoke the compiler, and the generated command file is saved for you to invoke or submit as a batch job.

The *file-spec* option allows you to enter a file specification for the generated command file. The default directory for the command file is the current default directory. By default, the program library manager provides a file name comprising up to the first 39 characters of the first unit name specified. If you use a wildcard character in the first unit name specified, the compiler uses the default name ACS_RECOMPILE. The default file type is .COM. No wildcard characters are allowed in the file specification.

By default, if you do not specify the /COMMAND qualifier, the program library manager deletes the generated command file when the RECOMPILE command completes normally or is terminated.

/CONFIRM
/NOCONFIRM (D)
Controls whether the RECOMPILE command asks you for confirmation before performing a possibly lengthy operation. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/COPY_SOURCE (D)
/NOCOPY_SOURCE
Controls whether a copied source file is created in the current program library when a compilation unit is recompiled without error. The ACS RECOMPILE command requires that a copied source file exist in the current program library; the ACS COMPILE command uses the copied source file if it cannot find an external source file when it is recompiling an obsolete unit or completing an incomplete generic instantiation (see Chapter 3). Copied source files may also be used by the VMS Debugger (see Chapter 6).

The /[NO]COPY_SOURCE qualifier has an effect with the RECOMPILE command only for those obsolete units in a parent library that are being recompiled into the current sublibrary to make them current. In this case, by default, a copied source file is created in the current program library when a unit is recompiled without error.

/DEBUG[=(option[,...])] (D)
/NODEBUG
Controls which debugger compiler options are provided. You can debug VAX Ada programs with the VMS Debugger (see Chapters 6 and 7). You can request the following options:

ALL	Provides both SYMBOLS and TRACEBACK
NONE	Provides neither SYMBOLS nor TRACEBACK
[NO]SYMBOLS	Controls whether debugger symbol records are included in the object file
[NO]TRACEBACK	Controls whether traceback information (a subset of the debugger symbol information) is included in the object file

By default, both debugger symbol records and traceback information are included in the object files (/DEBUG=ALL, or equivalently: /DEBUG).

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)
Controls whether a diagnostics file containing compiler messages and diagnostic information is created. The diagnostics file is supported only for use with Digital layered products, such as the VAX Language-Sensitive Editor.

By default, a diagnostics file is created from the copied source file for each unit that is recompiled.

/ERROR_LIMIT[=n]
/NOERROR_LIMIT
Controls whether execution of the RECOMPILE command for a given compilation unit is terminated upon the occurrence of the nth E-level error within that unit.

Error counts are not accumulated across a sequence of compilation units. If the /ERROR_LIMIT=n option is specified, each compilation unit may have up to n – 1 errors without terminating the compilation. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues.

The /ERROR_LIMIT=0 option is equivalent to ERROR_LIMIT=1.

By default, execution of the RECOMPILE command is terminated for a given compilation unit upon the occurrence of the 30th E-level error within that compilation unit (equivalent to /ERROR_LIMIT=30).

/KEEP (D)
/NOKEEP
Controls whether the batch log file generated is deleted after it is printed when the RECOMPILE command is executed in batch mode (the default mode).

By default, the log file is not deleted.

/LIST[=file-spec]
/NOLIST (D)
Controls whether a listing file is created. One listing file is created for each compilation unit (not file) recompiled by the RECOMPILE command. The default directory for listing files is the current default directory. The default file name of a listing file corresponds to the name of its compilation unit and uses the VAX Ada file-name conventions described in Chapter 1.

The default file type of a listing file is .LIS. No wildcard characters are allowed in the file specification.

By default, the RECOMPILE command does not create a listing file.

/LOG
/NOLOG (D)
Controls whether a list of all the units that must be recompiled is displayed.

By default, a list of the units that must be recompiled is not displayed.

/MACHINE_CODE
/NOMACHINE_CODE (D)
Controls whether generated machine code (approximating assembler notation) is included in the listing file.

By default, generated machine code is not included in the listing file.

/NAME=job-name
Specifies a string to be used as the job name and as the file name for the batch log file when the RECOMPILE command is executed in batch mode (the default mode). The job name can have from 1 to 39 characters.

By default, if you do not specify the /NAME qualifier, the program library manager creates a job name comprising up to the first 39 characters of the first unit name specified. If you do not specify the /NAME qualifier, but use a wildcard character in the first unit name specified, the compiler uses the default name ACS_RECOMPILE. In these cases, the job name is also the file name of the batch log file.

/NOTE_SOURCE (D)
/NONOTE_SOURCE
Controls whether the file specification of the source file is noted in the program library when a unit is recompiled without error. The COMPILE command uses this information to locate revised source files.

The /[NO]NOTE_SOURCE qualifier has no effect with the RECOMPILE command.

/NOTIFY (D)
/NONOTIFY
Controls whether a message is broadcast when the RECOMPILE command is executed in batch mode (the default mode). The message is broadcast to any terminal at which you are logged in, notifying you that your job has been completed or terminated.

By default, a message is broadcast.

/OPTIMIZE[=(option[,...])]
/NOOPTIMIZE
Controls the level of optimization that is applied in producing the compiled code. You can specify one of the following primary options:

TIME	Provides full optimization with time as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(SPACE) in the source code.
SPACE	Provides full optimization with space as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(TIME) in the source code.

DEVELOPMENT	Suggested when active development of a program is in progress. Provides some optimization, but development considerations and ease of debugging take preference over optimization. This option overrides pragmas that establish a dependence on a subprogram or generic body (the pragmas INLINE and INLINE_GENERIC), and thus reduces the need for recompilations when such bodies are modified. This option also disables generic code sharing.
NONE	Provides no optimization. Suppresses inline expansions of subprograms and generics, including those specified by the pragmas INLINE and INLINE_GENERIC. Suppresses occurrences of the pragma SHARE_GENERIC and disables generic code sharing.

The /NOOPTIMIZE qualifier is equivalent to /OPTIMIZE=NONE.

By default, the RECOMPILE command applies full optimization with time as the primary optimization criterion (like /OPTIMIZE=TIME, but observing uses of the pragma OPTIMIZE).

The /OPTIMIZE qualifier also has a set of secondary options that you can use separately or together with the primary options to override the default behavior for inline expansion (generic and subprogram) and generic code sharing.

The INLINE secondary option can have the following values (see the *VAX Ada Run-Time Reference Manual* for more information about inline expansion):

NONE	Disables subprogram and generic inline expansion. This option overrides any occurrences of the pragmas INLINE or INLINE_GENERIC in the source code, without your having to edit the source file. It also disables implicit inline expansion of subprograms. (*Implicit inline expansion* means that the compiler assumes a pragma INLINE for certain subprograms as an optimization.) A call to a subprogram or an instance of a generic in another unit is not expanded inline, regardless of the /OPTIMIZE options in effect when that unit was compiled.

NORMAL

Provides normal subprogram and generic inline expansion.

Subprograms to which an explicit pragma INLINE applies are expanded inline under certain conditions. In addition, some subprograms are implicitly expanded inline. The compiler assumes a pragma INLINE for calls to some small local subprograms (subprograms that are declared in the same unit as the unit in which the call occurs).

Instances are compiled separately from the unit in which the instantiation occurred unless a pragma INLINE_GENERIC applies to the instance. If a pragma INLINE_GENERIC applies and the generic body has been compiled, the generic is expanded inline at the point of instantiation.

SUBPROGRAMS

Provides maximal subprogram inline expansion and normal generic inline expansion.

In addition to the normal subprogram inline expansion that occurs when INLINE:NORMAL is specified, this option results in implicit inline expansion of some small subprograms declared in other units. The compiler assumes a pragma INLINE for any subprogram if it improves execution speed and reduces code size. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE were specified explicitly in the source code.

With this option, generic inline expansion occurs in the same manner as for INLINE:NORMAL.

GENERICS

Provides normal subprogram inline expansion and maximal generic inline expansion.

With this option, subprogram inline expansion occurs in the same manner as for INLINE:NORMAL.

The compiler assumes a pragma INLINE_GENERIC for every instantiation in the unit being compiled unless an explicit pragma SHARE_GENERIC applies or a generic body is not available. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE_GENERIC were specified explicitly in the source code.

MAXIMAL	Provides maximal subprogram and generic inline expansion. Maximal subprogram inline expansion occurs as for INLINE:SUBPROGRAMS, and maximal generic inline expansion occurs as for INLINE:GENERICS.

The SHARE secondary option can have the following values:

NONE	Disables generic sharing. This option overrides the effect of any occurrences of the pragma SHARE_GENERIC in the source code, without your having to edit the source file. In addition, instances do not share code from previous instantiations.
NORMAL	Provides normal generic sharing. Normally, the compiler will not attempt to generate shareable code for an instance (code that can be shared by subsequent instantiations) unless an explicit pragma SHARE_GENERIC applies to that instance. However, an instance will attempt to share code that resulted from a previous instantiation to which the pragma SHARE_GENERIC applied.
MAXIMAL	Provides maximal generic sharing. The compiler assumes that a pragma SHARE_GENERIC applies to every instance in the unit being compiled unless an explicit pragma INLINE_GENERIC applies. Thus, an instance will attempt to share code that resulted from a previous instantiation or to generate code that can be shared by subsequent instantiations. SHARE:MAXIMAL cannot be used in combination with INLINE:GENERICS or INLINE:MAXIMAL.

By default, the /OPTIMIZE qualifier primary options have the following secondary-option values:

/OPTIMIZE=TIME	=(INLINE:NORMAL, SHARE:NORMAL)
/OPTIMIZE=SPACE	=(INLINE:NORMAL, SHARE:NORMAL)
/OPTIMIZE=DEVELOPMENT	=(INLINE:NONE, SHARE:NONE)
/OPTIMIZE=NONE	=(INLINE:NONE, SHARE:NONE)

See Chapter 3 for more information about the /OPTIMIZE qualifier and its options.

/OUTPUT=file-spec
Requests that any program library manager output generated before the compiler is invoked be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the RECOMPILE command output is written to SYS$OUTPUT.

/PRELOAD
/NOPRELOAD (D)
Controls whether the RECOMPILE command processes copied source files in the same manner as the COMPILE/PRELOAD command processes external source files—to account for new compilation units or unit dependences. Because new units and unit dependences are normally not introduced by way of copied source files, the /PRELOAD qualifier has no effect when specified with the RECOMPILE command.

By default, the RECOMPILE command does not process copied source files to account for new compilation units or unit dependences.

/PRINTER[=queue-name]
/NOPRINTER (D)
Controls whether the batch job log file is queued for printing when the RECOMPILE command is executed in batch mode (the default mode).

The /PRINTER qualifier allows you to specify a particular print queue. The default print queue for the log file is SYS$PRINT.

By default, the log file is not queued for printing. If you specify the /NOPRINTER qualifier, the /KEEP qualifier is assumed.

/QUEUE=queue-name
Specifies the batch job queue in which the job is entered when the RECOMPILE command is executed in batch mode (the default mode).

By default, if the /QUEUE qualifier is not specified, the program library manager first checks whether the logical name ADA$BATCH is defined. If it is, the program library manager enters the job in the queue specified. Otherwise, the job is placed in the default system batch job queue, SYS$BATCH.

/SHOW[=option] (D)
/NOSHOW

Controls the listing file options included when a listing file is provided. You can specify one of the following options:

ALL	Provides all listing file options.
[NO]PORTABILITY	Controls whether a program portability summary is included in the listing file (see Chapter 5).
NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the RECOMPILE command provides a portability summary (/SHOW=PORTABILITY).

/SPECIFICATION_ONLY

Causes only the specifications of the units specified to be considered for recompilation. You can use the /CLOSURE qualifier with the /SPECIFICATION_ONLY qualifier to force only the specifications in the execution closure of the specified units to be considered for recompilation.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, all of the specifications, bodies, and subunits in the execution closure of the units specified are considered for compilation.

/SUBMIT

Directs the program library manager to submit the command file generated for the compiler to a batch queue. You can continue to enter commands in your current process without waiting for the batch job to complete. The compiler output is written to a log file.

By default, the program library manager submits the command file generated for the compiler to a batch queue.

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)
Controls whether the copied source file is to be checked only for correct syntax. If you specify the /SYNTAX_ONLY qualifier, other compiler checks are not performed (for example, semantic analysis, type checking, and so on), and the program library is not updated.

By default, the compiler performs all checks.

/WAIT
Directs the program library manager to execute the command file generated for the compiler in a subprocess. Execution of your current process is suspended until the subprocess completes. The compiler output is written directly to your terminal. Note that process logical names are propagated to the subprocess generated to execute the command file.

By default, the program library manager submits the command file generated for the compiler to a batch queue (by way of the RECOMPILE/SUBMIT command).

/WARNINGS[=(option[,...])]
/NOWARNINGS
Controls which categories of informational (I-level) and warning (W-level) messages are displayed and where those messages are displayed. You can specify any combination of the following message options:

WARNINGS: (*destination*[,...])
NOWARNINGS

WEAK_WARNINGS: (*destination*[,...])
NOWEAK_WARNINGS

SUPPLEMENTAL: (*destination*[,...])
NOSUPPLEMENTAL

COMPILATION_NOTES: (*destination*[,...])
NOCOMPILATION_NOTES

STATUS: (*destination*[,...])
NOSTATUS

The possible values of *destination* are ALL, NONE, or any combination of TERMINAL (terminal device), LISTING (listing file), and DIAGNOSTICS (diagnostics file). The message categories are summarized as follows (see Chapter 3 for more information):

WARNINGS	W-level: Indicates a definite problem in a legal program—for example, an unknown pragma.
WEAK_WARNINGS	I-level: Indicates a potential problem in a legal program—for example, a possible CONSTRAINT_ERROR at run time. These are the only kind of I-level messages that are counted in the summary statistics at the end of a compilation.
SUPPLEMENTAL	I-level: Additional information associated with preceding E-level or W-level diagnostics.
COMPILATION_NOTES	I-level: Information about how the compiler translated a program, such as record layout, parameter-passing mechanisms, or decisions made for the pragmas INLINE, INTERFACE, or the import-subprogram pragmas.
STATUS	I-level: End of compilation statistics and other messages.

The defaults are as follows:

```
/WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)
```

If you specify only some of the message categories with the /WARNINGS qualifier, the default values for the other categories are used.

Positional Qualifiers

/DATE_CHECK (D)
/NODATE_CHECK

Controls whether the RECOMPILE command checks the creation date and time of copied source files to determine which units in the closure of units specified are obsolete. If you specify the /NODATE_CHECK qualifier, the RECOMPILE command forces the recompilation of every unit specified, even though some units may not be obsolete; bodies and subunits of the specified units are also recompiled as necessary, to make them current. Entered units are not considered for recompilation when the /NODATE_CHECK qualifier is in effect.

If you specify the /NODATE_CHECK/CLOSURE qualifier, the RECOMPILE command forces the recompilation of every unit in the execution closure of the units specified.

You can use the /NODATE_CHECK qualifier to force the recompilation of a set of units using a particular combination of compiler qualifiers.

By default, the RECOMPILE command checks the creation date and time of copied source files (/DATE_CHECK), and recompiles only the copied source files for units that are obsolete.

/FORCE_BODY

Forces the recompilation of the bodies of the specified compilation units, regardless of whether or not they are obsolete.

The /FORCE_BODY qualifier can have different effects depending on its position in the command line and its interaction with other qualifiers:

- If you append the /FORCE_BODY qualifier to the RECOMPILE command string (as opposed to appending it to an individual unit parameter), the RECOMPILE command forces the recompilation of the bodies of each unit specified on the command line.
- If you append the /FORCE_BODY qualifier to an individual unit parameter, the RECOMPILE command forces the recompilation of the body of only that unit.
- If you specify the /FORCE_BODY qualifier with the /CLOSURE qualifier, the RECOMPILE command forces the recompilation of the bodies of all the units in the execution closure of the units specified.

By default, if the /FORCE_BODY qualifier is omitted, the specifications, bodies, and subunits of all of the units in the execution closure of the units specified are considered for recompilation.

Examples

1. ACS>

```
%I, The following units will be recompiled:
RESERVATIONS
    package specification           16-Apr-1989 13:34
    package body                    16-Apr-1989 13:34
SCREEN_IO
    package body                    16-Apr-1989 13:22
SCREEN_IO.INPUT
    procedure body                  16-Apr-1989 13:22
SCREEN_IO.INPUT.BUFFER
    function body                   16-Apr-1989 13:22
SCREEN_IO.OUTPUT
    procedure body                  16-Apr-1989 13:22
RESERVATIONS.RESERVE
    procedure body                  16-Apr-1989 13:35
RESERVATIONS.RESERVE.BILL
    procedure body                  16-Apr-1989 13:35
RESERVATIONS.CANCEL
    procedure body                  16-Apr-1989 13:36

%I, Job RESERVATIONS (queue ALL_BATCH, entry 180) started
    on FAST_BATCH
```

   Lists all of the units in the closure of unit RESERVATIONS that need to be recompiled, then submits the compiler command file generated by the program library manager as a batch job.

2. ACS>

   Creates and retains the compiler command file generated by the program library manager. The command file has the file name and type HOTEL.COM, by default. It contains commands to force (/NODATE_CHECK) the recompilation of all units in the closure (/CLOSURE) of unit HOTEL, with the /NOCHECK qualifier.

REENTER

Enters current references to units that were entered into the current program library and subsequently compiled in their original libraries.

Format

REENTER *unit-name[,...]*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/ENTERED=library	See text.
/[NO]LOCAL	/LOCAL
/[NO]LOG	/NOLOG

Positional Qualifiers	Defaults
/BODY_ONLY	See text.
/[NO]DATE_CHECK	/DATE_CHECK
/SPECIFICATION_ONLY	See text.

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units to be reentered into the current program library. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

REENTER

Description

The ACS REENTER command, like the ACS ENTER UNIT command, operates on a specified unit's specification plus its body and subunits, if any. For each unit specified, the REENTER command looks up the unit in its original program library and enters the current definition of the unit into the current program library. By default, if a specified unit's definition is current, it is not reentered.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)
Controls whether the REENTER command displays the unit name of each unit before reentering and requests you to confirm whether or not the unit should be reentered. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/ENTERED=library
Controls whether entered units are selected for reentering. You can use the library option to reenter units that were entered from a particular library. When you specify the /NOENTERED qualifier, only units that have been compiled or copied into the current program library are reentered. Note that when you specify the /ENTERED qualifier, local units are selected unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified are reentered from all of the libraries from which they were originally entered.

/LOG
/NOLOG (D)
Controls whether the name of a unit is displayed after it has been reentered.

By default, the names of reentered units are not displayed.

Positional Qualifiers

/BODY_ONLY
Reenters only the body of the specified unit.

When you append the /BODY_ONLY qualifier to the REENTER command string, any /SPECIFICATION_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION_ONLY qualifier to the REENTER command string or to the same unit name parameter.

By default, if the /BODY_ONLY qualifier is omitted, the specification, as well as the body, is reentered.

/DATE_CHECK (D)
/NODATE_CHECK
Controls whether the REENTER command compares the compilation date-time in the current program library and original library as the criterion for reentering a unit. If you specify the /NODATE_CHECK qualifier, the REENTER command will unconditionally reenter each unit specified in the command.

By default, the REENTER command compares the compilation date-time and reenters only those references that were obsolete.

/SPECIFICATION_ONLY
Reenters only the specification of the specified unit.

When you append the /SPECIFICATION_ONLY qualifier to the REENTER command string, any /BODY_ONLY qualifiers that are appended to parameters in the command line override the /BODY_ONLY qualifier for those particular parameters. You cannot append both the /SPECIFICATION_

ONLY qualifier and the /BODY_ONLY qualifier to the REENTER command string or to the same unit name parameter.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the body, as well as the specification, is reentered.

Examples

1.
```
ACS> REENTER/LOG *
%I, QUEUE_MANAGER entered
```

   Reenters every unit in the current program library that needs to be reentered, in this case the unit QUEUE_MANAGER.

2.
```
ACS> REENTER/NODATE_CHECK STACKS
```

   Unconditionally reenters the unit STACKS into the current program library, even if references to STACKS are current.

REORGANIZE

Optimizes the organization of the current VAX Ada program library (or the specified library).

NOTE

You can use this command only on a library to which you have exclusive access.

Format

REORGANIZE *[directory-spec]*

Command Qualifiers	Defaults
/[NO]LOG	/LOG
/OUTPUT=file-spec	See text.

Prompts

None.

Command Parameters

[directory-spec]
Specifies the VAX Ada program library to be reorganized. No wildcard characters are allowed in the directory specification.

If you do not specify a program library, the ACS REORGANIZE command reorganizes the current program library.

REORGANIZE

Description

The ACS REORGANIZE command optimizes the organization of the current program library or the specified library. You can use this command to improve the performance of any library; it is especially useful for improving the performance of libraries that have have been updated frequently.

To use the REORGANIZE command, you must have exclusive read-write access to the program library you are reorganizing. If another user is accessing the library when you enter the REORGANIZE command, the command will fail. One way to obtain exclusive access is to use the ACS SET LIBRARY/EXCLUSIVE command (note that this command will also fail if you cannot gain exclusive access when you enter it). You must enter the SET LIBRARY/EXCLUSIVE command interactively for it to have an effect.

Note that the SET LIBRARY/EXCLUSIVE command is not permitted for libraries across DECnet.

Command Qualifiers

/LOG (D)
/NOLOG
Controls whether a successful library reorganization is reported.

By default, a successful library reorganization is reported.

/OUTPUT=file-spec
Requests that the REORGANIZE command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the REORGANIZE command output is written to SYS$OUTPUT.

Example

```
ACS> REORGANIZE
%I, USER:[JONES.HOTEL.ADALIB] reorganized
```

Reorganizes the current program library (the library defined by the last ACS SET LIBRARY command). To determine when a library was last reorganized, enter the ACS SHOW LIBRARY/FULL command for that library.

SET LIBRARY

Defines a VAX Ada program library or program sublibrary as the current program library.

Format

SET LIBRARY *directory-spec*

Command Qualifiers	Defaults
/[NO]EXCLUSIVE	/NOEXCLUSIVE
/[NO]LOG	/LOG
/[NO]READ_ONLY	/NOREAD_ONLY

Prompts

_Library:

Command Parameters

directory-spec
Specifies the program library or program sublibrary that is to be defined as the current program library. For subsequent ACS commands to work, the specified directory must be a valid VAX Ada program library or program sublibrary, previously created with the CREATE LIBRARY or CREATE SUBLIBRARY command, respectively.

If you specify an invalid library, the SET LIBRARY command sets the library to whatever you specified (to prevent you from incorrectly modifying the wrong library).

Description

The ACS SET LIBRARY command establishes the current program library. VAX Ada units are compiled in the context of the current program library. The current program library is the target library for compiler output and for ACS commands in general.

The SET LIBRARY command performs the following steps:

1. Verifies that the specified directory is a valid VAX Ada program library or sublibrary. If the directory is invalid, an error message is issued.
2. Assigns the directory specification to the process logical name ADA$LIB. This assignment takes place even if the specified directory is invalid. The program library manager and the compiler use that logical name to maintain the current program library context when performing various operations.

The SET LIBRARY command does not affect the definition of the current default directory. The DCL SET DEFAULT command does not affect the definition of the current program library.

The /EXCLUSIVE and /READ_ONLY qualifiers are used for temporarily controlling access to program libraries in a shared library environment.

When using the SET LIBRARY command with the /EXCLUSIVE or /READ_ONLY qualifier values, you need to enter the command interactively (not as a DCL one-line command). For example:

```
ACS> SET LIBRARY/EXCLUSIVE [JONES.HOTEL.ADALIB]
```

When you use the /EXCLUSIVE or /READ_ONLY qualifier, the qualifier remains in effect until you exit from the program library manager or until another SET LIBRARY command is executed.

Command Qualifiers

/EXCLUSIVE
/NOEXCLUSIVE (D)
Controls whether the specified program library is opened for exclusive or shared (/NOEXCLUSIVE) access when the SET LIBRARY command is executed. Exclusive access to a compilation library over DECnet is not permitted.

If you execute a SET LIBRARY command without the /EXCLUSIVE qualifier or with the /NOEXCLUSIVE qualifier, then other processes are not denied access to the specified program library.

If you try to execute a SET LIBRARY/EXCLUSIVE command while the specified program library is being accessed by another process, the command will fail.

After executing a SET LIBRARY/EXCLUSIVE command, you have exclusive access to the specified program library until you exit from the program library manager or until another SET LIBRARY command is executed. Other processes are denied access to the program library until you exit from the program library manager or another SET LIBRARY command is executed.

By default, the SET LIBRARY command provides for shared (/NOEXCLUSIVE) access to the specified program library.

/LOG (D)
/NOLOG
Controls whether the program library directory specification of the library being set is displayed.

By default, the program library directory specification is displayed.

/READ_ONLY
/NOREAD_ONLY (D)
Controls whether the program library access is restricted to read-only access.

When you execute the SET LIBRARY/READ_ONLY command, the program library is opened only for reading for the duration of the ACS session. Therefore, you can only perform operations that do not modify the library: for example, ACS CHECK, DIRECTORY, EXPORT, EXTRACT SOURCE, LINK, SHOW LIBRARY, or SHOW PROGRAM. You can also copy and enter units from (not to) the library.

When you execute the SET LIBRARY/NOREAD_ONLY command, the program library is opened for reading, as well, but any subsequent command can try to open the library for a different kind of access.

By default, the /NOREAD_ONLY qualifier is in effect.

Examples

1.
```
ACS> SET LIBRARY [JONES.HOTEL.ADALIB]
%I, Current program library is USER:[JONES.HOTEL.ADALIB]
```

   Defines the program library [JONES.HOTEL.ADALIB], on the default device, as the current program library. The library is opened for both read and write access.

2.
```
ACS> SET LIBRARY/READ_ONLY DISK:[SMITH.SHARE.ADALIB]
%I, Current program library is DISK:[SMITH.SHARE.ADALIB]
```

   Defines the program library DISK:[SMITH.SHARE.ADALIB] as the current program library, with READ_ONLY access to the library.

SET PRAGMA

Redefines specified values of the program library characteristics LONG_ FLOAT, MEMORY_SIZE, and SYSTEM_NAME.

Note that use of this command may make units obsolete that depend on the previous value of a characteristic.

Format

SET PRAGMA

Command Qualifiers	Defaults
/LONG_FLOAT=option	See text.
/MEMORY_SIZE=n	See text.
/SYSTEM_NAME=system	See text.

Prompts

None.

Command Parameters

None.

Description

By default, a program library or sublibrary is created with the following system characteristics:

- LONG_FLOAT = G_FLOAT
- MEMORY_SIZE = 2147483647
- SYSTEM_NAME = VAX_VMS

These may be changed by compiling a unit that contains the pragmas LONG_FLOAT, MEMORY_SIZE, or SYSTEM_NAME.

The ACS SET PRAGMA command allows you to change the current program library's characteristics without having to compile a unit consisting of one of those pragmas.

The SET PRAGMA command may make units that depend on these characteristics obsolete. You can use the ACS RECOMPILE command to make obsolete units current.

Command Qualifiers

/LONG_FLOAT=option
Redefines the value of the program library characteristic LONG_FLOAT. The possible values are D_FLOAT and G_FLOAT.

By default, the current value of LONG_FLOAT is unchanged.

/MEMORY_SIZE=n
Redefines the value of the program library characteristic MEMORY_SIZE to n.

By default, the current value of MEMORY_SIZE is unchanged.

/SYSTEM_NAME=system
Redefines the value of the program library characteristic SYSTEM_NAME to a particular target operating system. The possible system values are VAX_VMS and VAXELN.

By default, the current value of SYSTEM_NAME is unchanged.

Example

```
ACS> SET PRAGMA/LONG_FLOAT=D_FLOAT
```

Redefines the current program library characteristic LONG_FLOAT to the value D_FLOAT.

SET SOURCE

Defines a source-file-directory search list for the ACS COMPILE command.

Format

SET SOURCE *directory-spec[,...]*

Prompts

_Search list:

Command Parameters

directory-spec[,...]
Specifies one or more VMS directories where the ACS COMPILE command should search for source files.

Description

The ACS COMPILE command searches the directories in the order specified in the ACS SET SOURCE command.

The search order takes precedence over the version number or revision date-time if different versions of a source file exist in two or more directories. Within any one directory, the version of a particular file that has the highest number is considered for compilation.

The search list specified by SET SOURCE remains in effect until another SET SOURCE command is executed, or until the process logs out.

If no SET SOURCE command is executed, the default search order is as follows:

1. SYS$DISK:[ ] (the current default directory)
2. ;0 (the directory that contained the file when it was last compiled), or node::;0 (if the file specification of the source file being compiled contains a node name)

Examples

1. ```
   ACS> SET SOURCE SYS$DISK:[],USER:[JONES.HOTEL],;0
   ```

   Defines the source-file search list to be: first, the current default directory (SYS$DISK:[ ]); second, the directory USER:[JONES.HOTEL]; third, the directory where the particular source file was last compiled (;0).

2. ```
   ACS> SET SOURCE SYS$DISK:[],CMS$LIB
   ```

   Defines the source-file search list to be: first, the current default directory (SYS$DISK:[ ]); second the current CMS library, as defined by the most recent CMS SET LIBRARY command, which defines the logical name CMS$LIB.

SHOW LIBRARY

Displays information about one or more VAX Ada program libraries, including directory specifications, library characteristics, and units defined in each library.

Format

SHOW LIBRARY *[directory-spec[,...]]*

Command Qualifiers	Defaults
/BODY_ONLY	See text.
/BRIEF	See text.
/[NO]ENTERED[=library]	/ENTERED
/FULL	See text.
/[NO]LOCAL	/LOCAL
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/SPECIFICATION_ONLY	See text.
/UNITS	See text.

Prompts

None.

Command Parameters

[directory-spec[,...]]
Specifies one or more VAX Ada program libraries for display. No wildcard characters are allowed in the directory specifications.

If you do not specify a program library, the SHOW LIBRARY command displays information about the current program library.

Description

The ACS SHOW LIBRARY command displays various information about one or more specified program libraries, including the full directory specifications, library characteristics, and units defined in each program library.

The output of the SHOW LIBRARY command depends on whether the /UNITS qualifier is used and, in addition, whether the /BRIEF or /FULL formatting qualifier is used.

If you do not specify a qualifier, the SHOW LIBRARY command displays the directory specifications of the program libraries specified.

Command Qualifiers

/BODY_ONLY
Displays only the bodies of the specified units when you use the /UNITS qualifier.

You cannot append both the /BODY_ONLY qualifier and the /SPECIFICATION ONLY qualifier to the SHOW LIBRARY/UNITS command string.

By default, if the /BODY_ONLY qualifier is omitted, the specifications, as well as the bodies, are displayed.

/BRIEF
Displays the program library directory specifications.

If used with the /UNITS qualifier, also lists the names of all units contained in each program library.

/ENTERED[=library] (D)
/NOENTERED
Controls whether entered units are displayed when you use the /UNITS qualifier. You can use the library option to display units that were entered from a particular library. When you specify the /NOENTERED qualifier, only units that have been compiled or copied into the current program library are displayed. Note that when you specify the /ENTERED qualifier, local units are displayed unless the /NOLOCAL qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units, as well as entered units are displayed when you use the /UNITS qualifier.

/FULL
Displays, for each program library specified, the directory specifications and the values of the program library characteristics LONG_FLOAT, MEMORY_SIZE, and SYSTEM_NAME.

If a program sublibrary is specified, identifies the parent library.

If used with the /UNITS qualifier, also displays, for each program library specified, each unit's name, kind, compilation date-time, and the file specifications of the files associated with each unit.

/LOCAL (D)
/NOLOCAL
Controls whether local units (those units that were added to the library by a compilation or a COPY UNIT command) are displayed when you use the /UNITS qualifier. Note that when you specify the /LOCAL qualifier, entered units are displayed unless the /NOENTERED qualifier is also in effect (the defaults for these qualifiers are /LOCAL and /ENTERED).

By default, all units specified, including local units, are displayed.

/OUTPUT=file-spec
Requests that the SHOW LIBRARY command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the SHOW LIBRARY command output is written to SYS$OUTPUT.

/SPECIFICATION_ONLY
Displays only the specifications of the specified units when you use the /UNITS qualifier.

You cannot append both the /SPECIFICATION_ONLY qualifier and the /BODY_ONLY qualifier to the SHOW LIBRARY/UNITS command string.

By default, if the /SPECIFICATION_ONLY qualifier is omitted, the bodies, as well as the specifications, are displayed.

/UNITS

Lists each unit that is defined in the specified program libraries. The level of information displayed depends on whether the /BRIEF or /FULL qualifier is also used. The unit information displayed is identical to that displayed by the DIRECTORY command.

Examples

1.
```
ACS> SHOW LIBRARY
%I, Current program library is USER:[JONES.HOTEL.ADALIB]
```

   Identifies the current program library.

2.
```
ACS> SHOW LIBRARY/FULL [JONES.HOTEL.SUBLIB]

Program library USER:[JONES.HOTEL.SUBLIB]

    Sublibrary
        of USER:[HOTEL.ADALIB]

Created:          15-Apr-1989 14:44, by VAX Ada 2.0
Last reorganized: 16-Apr-1989 13:40

Pragmas that affect STANDARD and SYSTEM:

    pragma LONG_FLOAT(D_FLOAT)
    pragma MEMORY_SIZE(2147483647)
    pragma SYSTEM_NAME(VAX_VMS)
```

   Identifies USER:[JONES.HOTEL.SUBLIB] as a sublibrary of USER:[HOTEL.ADALIB].

SHOW PROGRAM

Displays information about the execution closure of one or more units in the current program library.

Format

SHOW PROGRAM *unit-name[,...]*

Command Qualifiers	Defaults
/BRIEF	See text.
/FULL	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PORTABILITY	/NOPORTABILITY

Prompts

_Unit:

Command Parameters

unit-name[,...]
Specifies one or more units, in the current program library, about whose execution closure various information is to be shown. You must express subunit names using selected component notation as follows:

```
ancestor-unit-name[.parent-unit-name[...]].subunit-name
```

The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for more information on wildcard characters.)

Description

The ACS SHOW PROGRAM command displays information about all of the units in the execution closure of the specified units.

Units are listed by name in alphabetical order. Subunit names are shown using selected component notation.

The output of the SHOW PROGRAM command depends on whether the /BRIEF, /FULL, or no formatting qualifier is used.

If you do not specify a qualifier, the SHOW PROGRAM command displays a level of information that is part way between that displayed with the /BRIEF and /FULL qualifiers.

If you do not specify a qualifier, the SHOW PROGRAM command displays the information provided by the /BRIEF qualifier plus the following information for each unit in the closure:

- The **with** list of that unit
- The duration specified with the pragma TIME_SLICE
- The names of units mentioned in one or more ELABORATE pragmas for that unit
- The names of units that the unit has established a dependence on as a result of subprogram inline expansion
- The names of units that the unit has established a dependence on as a result of generic inline expansion

Command Qualifiers

/BRIEF

Displays the following information:

- The directory specification of the current program library.
- The values of the program library characteristics LONG_FLOAT, MEMORY_SIZE, and SYSTEM_NAME.
- For each unit in the closure of the specified units: the unit name; the kind of unit (for example, procedure body); the date and time of the last compilation; and the file specification of the source file, or (if the unit

was entered into the current program library) the directory specification of the other library.

/FULL
Displays the information provided by the SHOW PROGRAM command when used with no qualifier plus, for each unit in the closure, the file specifications of the associated files.

/OUTPUT=file-spec
Requests that the SHOW PROGRAM command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. The default file type is .LIS. If you specify a file type but omit the file name, the default file name is ACS. No wildcard characters are allowed in the file specification.

By default, the SHOW PROGRAM command output is written to SYS$OUTPUT.

/PORTABILITY
/NOPORTABILITY (D)
Lists, for the closure of the specified units, a portability summary indicating use of potentially nonportable features. For example:

- Pragmas
- VMS predefined floating-point types
- Enumeration representation clauses

Implementation-defined features are flagged with an asterisk (*).

See Chapter 3 for a discussion of portability.

Example

```
ACS> SHOW PROGRAM/PORTABILITY ADA_CALLER

ADA_CALLER
16-Apr-1989 13:45

Program library USER:[PROJ.ADALIB]

Created:          15-Apr-1989 14:44, by VAX Ada 2.0
Last reorganized: 16-Apr-1989 13:40

Pragmas that affect STANDARD and SYSTEM:

    pragma LONG_FLOAT(G_FLOAT)
    pragma MEMORY_SIZE(2147483647)
    pragma SYSTEM_NAME(VAX_VMS)

The closure of the specified units is:

ADA_CALLER
   Procedure body
      Compiled:       16-Apr-1989 11:26
      Source file:    31-Jul-1987 16:23  USER:[TEST]ADA_CALLER.ADA;2
      With list:      SQR
                      INTEGER_TEXT_IO

INTEGER_TEXT_IO
   Package instantiation
      Compiled:       13-Apr-1989 23:38
      Entered from:   SYS$COMMON:[SYSLIB.ADALIB]
      With list:      TEXT_IO
 .
 .
 .
IO_EXCEPTIONS
   Package specification
      Compiled:       13-Apr-1989 23:35
      Entered from:   SYS$COMMON:[SYSLIB.ADALIB]

SQR
   Function specification
      Compiled:       16-Apr-1989 11:24
      Source file:    31-Jul-1987 16:21  USER:[TEST]SQR_.ADA;2

   Foreign function body
      Object file:    16-Apr-1989 11:25  SQR.OBJ;1
```

SHOW PROGRAM

```
SYSTEM
   Builtin package

TEXT_IO
   Package specification
      Compiled:        13-Apr-1989 23:37
      Entered from:    SYS$COMMON:[SYSLIB.ADALIB]
      With list:       IO_EXCEPTIONS

   Package body
      Compiled:        13-Apr-1989 23:37
      Entered from:    SYS$COMMON:[SYSLIB.ADALIB]
      With list:       SYSTEM


PORTABILITY SUMMARY

predefined SHORT_INTEGER or SHORT_SHORT_INTEGER
                        SYSTEM spec

with SYSTEM             TEXT_IO body

predefined F_FLOAT, D_FLOAT, G_FLOAT or H_FLOAT*
                        TEXT_IO body

enumeration representation clause
                        SYSTEM spec
                        TEXT_IO spec

length SIZE representation clause
                        SYSTEM spec

record representation clause
                        SYSTEM spec

pragma PACK             SYSTEM spec

pragma IMPORT_EXCEPTION*
                        IO_EXCEPTIONS spec

pragma IMPORT_FUNCTION* SQR spec
                        TEXT_IO spec

pragma IMPORT_PROCEDURE*
                        TEXT_IO

pragma INTERFACE        SQR spec
                        TEXT_IO

         where * indicates an implementation-defined feature
```

Displays information about the closure of the unit ADA_CALLER, which also includes the unit SQR and a number of VAX Ada predefined units.

The /PORTABILITY qualifier produces a portability summary for the units displayed. The unit display and portability summary indicate that the body of SQR was copied into the current program library, USER:[PROJ.ADALIB], as a foreign body (file SQR.OBJ).

SHOW SOURCE

Displays the source-file-directory search list used by the ACS COMPILE command.

Format

SHOW SOURCE

Prompts

None.

Command Parameters

None.

Description

The ACS SHOW SOURCE command displays the directory list specified in the last ACS SET SOURCE command. See the description of the SET SOURCE command.

Example

```
ACS> SHOW SOURCE
%I, Current source search list (ADA$SOURCE) is
        USER:[JONES.HOTEL]
        DISK:[SMITH.SHARE]
```

Shows that the directories to be searched by the ACS COMPILE command for external source files are first the directory USER:[JONES.HOTEL] and then the directory DISK:[SMITH.SHARE].

SHOW VERSION

Displays the version of VAX Ada that is installed on your system.

Format

SHOW VERSION

Prompts

None.

Command Parameters

None.

Description

The ACS SHOW VERSION command displays a string that gives the version number of VAX Ada (compiler and program library manager) that is installed on your system.

Example

```
ACS> SHOW VERSION
VAX Ada V2.0-0
```

Shows that Version 2.0 of VAX Ada is currently running on the user's system.

SPAWN

Creates a subprocess of the current process and suspends execution of the current process.

Format

SPAWN *[DCL-command]*

Prompts

None.

Command Parameters

[DCL-command]
Specifies an optional DCL command.

Description

The ACS SPAWN command creates a subprocess of the current process and suspends execution of the current process.

If you specify a DCL command, that command is executed in a subprocess, and control is returned to the program library manager after the command is executed.

If you do not specify a DCL command, an interactive subprocess is created allowing you to execute a whole series of DCL commands interactively. You can return to the program library manager by logging out of the subprocess (by entering a DCL LOGOUT command) or entering a DCL ATTACH command. See the description of the DCL ATTACH command in the *VMS DCL Dictionary*.

Example

```
ACS> SPAWN MAIL        ! from process JONES
MAIL>
  .
  .
  .
MAIL> ATTACH JONES
%I, Control returned to process JONES
ACS>
  .
  .
  .
ACS> ATTACH JONES_1
MAIL>
```

The ACS SPAWN MAIL command, entered from process JONES, invokes the VMS Mail Utility in a subprocess named JONES_1. The DCL ATTACH command entered from MAIL (subprocess JONES_1) returns control back to process JONES. The ACS ATTACH command entered interactively from the program library manager (process JONES) switches control back to subprocess JONES_1.

VERIFY

Performs a series of consistency checks on the current program library (or the specified library) to determine whether the library structure and library files are in valid form. The ACS VERIFY command optionally corrects some of the inconsistencies detected.

Format

VERIFY *[directory-spec]*

Command Qualifiers	Defaults
/[NO]CONFIRM	/NOCONFIRM
/[NO]LOG	/NOLOG
/OUTPUT=file-spec	See text.
/[NO]REPAIR	/NOREPAIR

Prompts

None.

Command Parameters

directory-spec
Specifies the VAX Ada program library to be verified. No wildcard characters are allowed in the directory specification.

If you do not specify a program library, the ACS VERIFY command verifies the current program library.

Description

The ACS VERIFY command checks the following items (unless otherwise stated, only files in the specified program library are checked):

- The format of the library index file.
- Whether all files cataloged in the library index file exist in the program library and are accessible—that is, all object (.OBJ), compilation unit (.ACU), and copied source (.ADC) files. In the case of entered units, the VERIFY command checks whether the files exist in the library from which they were entered.
- Whether all .OBJ, .ACU, and .ADC files that exist in the program library directory are cataloged in the library index file.
- Whether temporary files used by the REORGANIZE command are in the program library.
- The format of the compilation unit files (.ACU).
- Whether the protection code of cataloged .OBJ, .ACU, and .ADC files is consistent with that of the library index file (see Chapter 5).

If inconsistencies are found, the VERIFY command issues error messages indicating the units or files that are erroneous.

The kinds of inconsistencies detected by the VERIFY command are typically not detected by the ACS CHECK command, which is used to determine whether any units in a closure are missing or obsolete.

You can use the /REPAIR qualifier to correct some of the inconsistencies reported by the VERIFY command. When the /REPAIR qualifier is used, the VERIFY command performs the same checks as when the qualifier is not used, but corrective action is taken only on the specified program library or, by default, on the current program library. No corrective action is taken for entered units.

Command Qualifiers

/CONFIRM
/NOCONFIRM (D)
Controls whether the VERIFY/REPAIR command asks for confirmation before deleting unit index entries from the library index file, or deleting

uncataloged files from the program library directory. If you specify the /CONFIRM qualifier, the possible responses are as follows:

- Affirmative responses are YES, TRUE, and 1.
- Negative responses are NO, FALSE, 0, and the RETURN key.
- QUIT or CTRL/Z indicates that you want to stop processing the command at that point.
- ALL indicates that you want to continue processing the command without any further prompts.

You can use any combination of upper- and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, Y, YE, or YES). If you type a response other than one of those in the list, the prompt is reissued.

By default, no confirmation is requested.

/LOG
/NOLOG (D)
Controls whether the name of a unit or the specification of a file is displayed as that unit or file is verified.

By default, the names of units or files being verified are not displayed.

/OUTPUT=file-spec
Requests that the VERIFY command output be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is ACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the VERIFY command output is written to SYS$OUTPUT.

/REPAIR
/NOREPAIR (D)
Controls whether the VERIFY command repairs some of the inconsistencies that it has detected.

To use the /REPAIR qualifier, you must have exclusive read-write access to the program library you are repairing. If another user is accessing the library when you enter the VERIFY/REPAIR command, the command will fail. One way to obtain exclusive access is to use the ACS SET LIBRARY/EXCLUSIVE command (note that this command will also fail if you cannot gain exclusive access when you enter it). You must enter the SET LIBRARY/EXCLUSIVE command interactively for it to have an effect.

Note that the SET LIBRARY/EXCLUSIVE command is not permitted for program libraries over DECnet.

The VERIFY/REPAIR command takes the following actions:

- Identifies any files in the program library directory that are not cataloged in the library index file. Deletes any uncataloged files with a file type of .OBJ, .ACU, or .ADC. Deletes any temporary files remaining from an interrupted ACS REORGANIZE command. Deletes any other uncataloged files if you have also specified the /CONFIRM qualifier and given an affirmative response.
- As necessary, changes the file protection on .OBJ, .ACU, and .ADC files to be consistent with the protection code for the library index file.
- Marks as obsolete any unit whose .OBJ or .ACU file is inaccessible. A later VERIFY/REPAIR command will reset any such marks if the associated files are again available.
- Removes references to inaccessible copied source files (.ADC) from the library index file.
- Deletes any index entry with an illegal format from the library index file.

By default, the VERIFY command only checks for inconsistencies and takes no corrective action.

Examples

1.
```
ACS> VERIFY
%I, USER:[JONES.HOTEL.ADALIB] verified
```

   Checks the current program library. No inconsistencies have been detected.

2.
```
ACS> SET LIBRARY/EXCLUSIVE [PROJ.ADALIB]
%I, Current program library is USER:[PROJ.ADALIB]
ACS> VERIFY/REPAIR/LOG
  .
  .
  .
%I, STARLET verified
%I, STR verified
%E, Inconsistent file protection [PROJ.ADALIB]SQR.OBJ;1
%W, SQR verified and repaired
  .
  .
  .
%E, Error opening [PROJ.ADALIB]TEST_STACKS.OBJ;2 as input
-E, file not found
%W, TEST_STACKS verified and repaired
  .
  .
  .
%I, Units with inaccessible files are obsolete. If repair
    (VERIFY/REPAIR) is not possible, then recompilation of
    these units is necessary; after entering a VERIFY/REPAIR
    command, the CHECK command will show any obsolete units
%W, USER:[PROJ.ADALIB] verified and repaired
ACS> RECOMPILE TEST_STACKS
```

Defines the program library [PROJ.ADALIB] as the current program library, with exclusive read-write access. This step is necessary before using the VERIFY/REPAIR command.

The VERIFY/REPAIR command then notes that the protection of file SQR.OBJ is inconsistent with that of the library index file and changes the protection to make it consistent; marks the unit TEST_STACKS as obsolete, because its .OBJ file (TEST_STACKS.OBJ;2) is inaccessible; and issues a summary message that the program library has been verified and repaired.

The RECOMPILE command then makes the obsolete unit, TEST_ STACKS, current.

Appendix B

Debugger Command Summary

This appendix lists all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

During a debugging session, you can get online HELP on any command and its qualifiers by typing the HELP command followed by the name of the command in question. The HELP command has the following form:

```
HELP command
```

B.1 Starting and Terminating a Debugging Session

($) RUN[1]	Invokes the debugger if ACS LINK/DEBUG was used
($) RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed
CTRL/Z or EXIT	Ends a debugging session, executing all exit handlers
QUIT	Ends a debugging session without executing any exit handlers declared in the program
CTRL/Y	Interrupts a debugging session and returns you to DCL level
CTRL/C	Has the same effect as CTRL/Y, unless the program has a CTRL/C service routine
($) CONTINUE[1]	Resumes a debugging session after a CTRL/Y interruption

[1]This is a DCL command, not a debugger command.

($) DEBUG[1]	Resumes a debugging session after a CTRL/Y interruption but returns you to the debugger prompt
ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL ATTACH command)
SPAWN	Creates a subprocess; allows you to enter DCL commands without interrupting your debugging context (similar to the DCL SPAWN command)

[1]This is a DCL command, not a debugger command.

B.2 Controlling and Monitoring Program Execution

GO	Starts or resumes program execution
STEP	Executes the program up to the next line, instruction, or specified instruction
{ SET | SHOW } STEP	Establishes or displays the default qualifiers for the STEP command
{ SET | SHOW | CANCEL } BREAK	Sets, displays, or cancels breakpoints
{ SET | SHOW | CANCEL } TRACE	Sets, displays, or cancels tracepoints
{ SET | SHOW | CANCEL } WATCH	Sets, displays, or cancels watchpoints
SHOW CALLS	Identifies the currently active routine calls
SHOW STACK	Gives additional information about the currently active routine calls
CALL	Calls a routine

B.3 Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location

SET MODE [NO]OPERANDS	Controls whether, when you examine an instruction, the address and contents of the instruction operands are displayed
DEPOSIT	Changes the value of a variable or the contents of a program location
EVALUATE	Evaluates a language or address expression

B.4 Controlling Type Selection and Symbolization

{ SET / SHOW / CANCEL } RADIX	Establishes the radix for data entry and display, displays the radix, or restores the radix
{ SET / SHOW / CANCEL } TYPE	Establishes the type to be associated with untyped program locations, displays the type, or restores the type
SET MODE [NO]G_FLOAT	Controls whether double-precision, floating-point constants are interpreted as G_FLOAT or D_FLOAT
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name plus byte offset
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses
SYMBOLIZE	Converts a virtual address to a symbolic address

B.5 Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program
{ SET / SHOW / CANCEL } MODULE	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module (in Ada, a module corresponds to a compilation unit)
{ SET / SHOW / CANCEL } IMAGE	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image

SET MODE [NO]DYNAMIC	Controls whether modules and shareable images are automatically set when the debugger interrupts execution
{ SET / SHOW / CANCEL } SCOPE	Establishes, displays, or restores the scope for symbol lookup

B.6 Displaying Source Code

TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
{ SET / SHOW / CANCEL } SOURCE	Creates, displays, or cancels a source directory search list
SEARCH	Searches the source code for the specified string
{ SET / SHOW } SEARCH	Establishes or displays the default qualifiers for the debugger SEARCH command
{ SET / SHOW } MAX_SOURCE_FILES	Establishes or displays the maximum number of source files that may be kept open at one time
{ SET / SHOW } MARGINS	Establishes or displays the left and right margin settings for displaying source code

B.7 Using Screen Mode

SET MODE [NO]SCREEN	Enables or disables screen mode
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
DISPLAY	Modifies an existing display
{ SET / SHOW / CANCEL } DISPLAY	Creates, identifies, or deletes a display
{ SET / SHOW / CANCEL } WINDOW	Creates, identifies, or deletes a window definition
SELECT	Selects a display for a display attribute

SHOW SELECT	Identifies the displays selected for each of the display attributes
SCROLL	Scrolls a display
SAVE	Saves the current contents of a display and writes it to another display
EXTRACT	Saves a display or the current screen state and writes it to a file
EXPAND	Expands or contracts a display
MOVE	Moves a display across the screen
{ SET / SHOW } TERMINAL	Establishes or displays the height and width of the screen
CTRL/W or DISPLAY/REFRESH	Refreshes the screen

B.8 Editing Source Code

EDIT	Invokes an editor during a debugging session
{ SET / SHOW } EDITOR	Establishes or identifies the editor invoked by the debugger EDIT command

B.9 Defining Symbols

DEFINE	Defines a symbol as an address, command, or value
DELETE	Deletes symbol definitions
{ SET / SHOW } DEFINE	Establishes or displays the default qualifier for the debugger DEFINE command
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined

B.10 Using Keypad Mode

SET MODE [NO]KEYPAD	Enables or disables keypad mode
DEFINE/KEY	Creates key definitions

DELETE/KEY	Deletes key definitions
SET KEY	Establishes the key definition state
SHOW KEY	Displays key definitions

B.11 Using Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures
{ SET / SHOW } LOG	Specifies or identifies the debugger log file
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed
SHOW OUTPUT	Displays the current output options established by the debugger SET OUTPUT command
{ SET / SHOW } ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures
@file-spec	Executes a command procedure

B.12 Using Control Structures

IF	Executes a list of commands conditionally
FOR	Executes a list of commands repetitively
REPEAT	Executes a list of commands repetitively
WHILE	Executes a list of commands conditionally
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop

B.13 Additional Commands

SET PROMPT	Specifies the debugger prompt
SET OUTPUT [NO]TERMINAL	Controls whether debugger output is displayed or suppressed, except for diagnostic messages
{ SET / SHOW } LANGUAGE	Establishes or displays the current programming language
{ SET / SHOW } EVENT_FACILITY	Establishes or identifies the current run-time facility for language-specific events
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program
{ SET / SHOW } TASK	Modifies the tasking environment or displays task information
{ DISABLE / ENABLE / SHOW } AST	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled
SET MODE [NO]SEPARATE	Controls whether a separate window is created on a VAXstation for debugger input and output (this command has no effect on VT-series terminals)

Appendix C

Using VAX Ada with the VAX Language-Sensitive Editor and Source Code Analyzer

This appendix provides overviews of the VAX Language-Sensitive Editor (LSE) and VAX Source Code Analyzer (SCA) tools and explains how you can use them with VAX Ada.

These tools are not included with the VAX Ada software; you must purchase them separately. For information on how to purchase them, contact your Digital sales representative.

C.1 Using VAX Ada with LSE

LSE is a powerful and flexible text editor designed specifically for software development. In addition to text-editing features, LSE provides the following software development features:

- Formatted language constructs, or templates, for most VAX programming languages, including VAX Ada. These templates include the keywords and punctuation used in source programs, and use placeholders to indicate locations in the source code where additional program text must be provided.
- Extensive online HELP for language constructs, as well as for LSE commands and key definitions.
- The ability to compile, review, and correct compilation errors from within the editor.

- Integration with the VAX Source Code Analyzer (SCA) and VAX DEC/Code Management System (CMS). SCA and CMS commands can be entered from within the editor to make program analysis and source file management more efficient.

The following sections describe some of the key features of LSE, and give Ada-specific information where appropriate. For more details on advanced features of LSE, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

C.1.1 Starting and Ending an Editing Sesssion

To invoke LSE, enter the LSEDIT command at the DCL prompt, specifying a file name with the .ADA file type as a parameter. For example:

```
$ LSEDIT HOTEL.ADA
```

To end an LSE session, press CTRL/Z to get the LSE> prompt. If you wish to save modifications to your file, enter the EXIT command. If you do not wish to save the file or any modification to the file, enter the QUIT command.

C.1.2 Obtaining Help

You can obtain both LSE help and Ada language help at any time during your editing session, as follows:

- To obtain a diagram of the keypad and LSE key bindings, press the HELP key (PF2).
- To obtain a listing of the keypad keys and their descriptions, press CTRL/Z to get the LSE> prompt, and type the SHOW KEY command.
- To obtain a list of LSE commands and their explanations, press CTRL/Z to get the LSE> prompt, and type HELP COMMANDS.
- To display a list of all of the predefined tokens or placeholders for the language of the current buffer, press CTRL/Z to get the LSE> prompt, and type the SHOW TOKEN or SHOW PLACEHOLDER command.
- To obtain language-specific help on a particular keyword or placeholder, position the cursor on the keyword or placeholder and press PF1-PF2. Help is not available for all keywords and placeholders.

C.1.3 Entering Source Code Using Tokens and Placeholders

To help you enter syntactically correct source code, LSE provides tokens and placeholders. These are language elements that have been predefined for Ada (as well as for other LSE-supported languages). You expand tokens and placeholders into templates for Ada language constructs.

Tokens are Ada reserved words or other keywords that you type into your editing buffer and then expand using CTRL/E or the EXPAND command. The expansions provide templates for the corresponding language constructs. For example, you can type the reserved word **if** into your editing buffer, press CTRL/E, and obtain the following template:

```
if {condition} then
    {statement}...
[elsif_part]...
[else_part]
end if;
```

Placeholders are markers for places in the source code where you must provide additional program text or choose from a number of options. Placeholders are inserted into the editing buffer by LSE, as the result of expanding other placeholders or tokens. In the preceding example, {condition}, {statement}, [elsif_part], and [else_part] are placeholders. Unlike tokens, you cannot type in and expand placeholders.

Placeholders can be either required or optional. Required placeholders are delimited by braces (for example, {condition}); they represent places in the source code where you must provide program text. Optional placeholders are delimited by brackets (for example, [else_part]); they represent places in the source code where you can either provide additional constructs or erase the placeholder.

Placeholders followed by a horizontal ellipsis indicate that more than one of the represented items is appropriate. A placeholder of the form [placeholder]... indicates that zero or more items are appropriate; a placeholder of the form {placeholder}... indicates that one or more items are appropriate.

You can expand, erase, or type directly over placeholders. When you type over a placeholder, the placeholder is automatically removed, and the text you type is inserted into the buffer. If more than one item represented by a placeholder is appropriate, LSE continues to provide a placeholder until you press CTRL/K (to erase the remaining placeholder).

When you expand a placeholder (using CTRL/E or the EXPAND command), one of three events occurs:

- The placeholder is automatically replaced with a template of language constructs. This kind of placeholder is called a *nonterminal* placeholder.
- Text appears in a separate window to help you supply a value. This kind of placeholder is called a *terminal* placeholder. When you supply a value or press the spacebar, the window disappears.
- A menu appears in a separate window to provide you with options that you can select and expand into templates. This kind of placeholder is called a *menu* placeholder. When you choose an option, the window disappears and the placeholder is replaced with another template. The window also disappears when you press the spacebar.

You can write a complete program by repeatedly expanding templates until you reach terminal placeholders. You can also type in values or constructs at a higher level.

LSE provides two ways to enter commands: keypad mode and command-line mode. When you invoke LSE, you are in keypad mode, and the text that you type is inserted into a buffer. To get to command-line mode, use one of the following procedures:

- Press the DO key or COMMAND key (PF1-7). The LSE Command> prompt appears at the bottom of the screen. After you enter one line-mode command, you are automatically returned to keypad mode.
- Press CTRL/Z. The LSE> prompt appears at the bottom of the screen. After you enter as many line-mode commands as you like, press CTRL/Z again or enter the CONTINUE command to return to keypad mode.

Table C–1 lists the LSE commands for manipulating tokens and placeholders.

Table C–1: VAX LSE Commands for Manipulating Tokens and Placeholders

Command	Key Binding	Function
EXPAND	CTRL/E	Expands a placeholder
UNEXPAND	PF1-CTRL/E	Reverses the effect of the most recent placeholder expansion
GOTO PLACEHOLDER/FORWARD	CTRL/N	Moves the cursor to the next placeholder
GOTO PLACEHOLDER/REVERSE	CTRL/P	Moves the cursor to the previous placeholder
ERASE PLACEHOLDER/FORWARD	CTRL/K	Erases a placeholder
UNERASE PLACEHOLDER	PF1-CTRL/K	Restores the most recently erased placeholder
None	Down arrow	Moves the indicator down through a menu
None	Up arrow	Moves the indicator up through a menu
None	{ ENTER RETURN }	Selects a menu option

You can use the SHOW TOKEN and SHOW PLACEHOLDER commands to display a list of all defined tokens and placeholders, or a particular token or placeholder. To copy the listed information into a separate file, first enter the appropriate SHOW command to put the list into the $SHOW buffer. Then enter the following command:

```
LSE>  WRITE/BUFFER=$SHOW filename
```

To obtain a hard copy of the list, use the PRINT command at DCL level to print the file you created.

C.1.4 Compiling and Reviewing Source Code

The LSE COMPILE and REVIEW commands allow you to compile your code and review compilation errors without leaving your editing session.

The LSE COMPILE command enters a DCL command in a subprocess to invoke the VAX Ada compiler. Remember that you must have created and defined a VAX Ada program library before you can compile Ada source

code using the DCL ADA command (see Chapters 1, 2, and 3 for more information on program libraries and compilation).

NOTE

The LSE COMPILE command is equivalent to the DCL ADA command, not the ACS COMPILE command.

By default, the LSE COMPILE command executes the DCL ADA/DIAGNOSTICS command (including any default qualifiers for the DCL ADA command, such as /DEBUG, /OPTIMIZE, and /NOANALYSIS_DATA). The /DIAGNOSTICS qualifier causes the Ada compiler to generate a file (.DIA) of compilation diagnostics, which you can review with the LSE REVIEW command. For example, if you enter the LSE COMPILE command while you are in the buffer HOTEL.ADA, the following DCL command is executed:

```
$ ADA HOTEL.ADA/DIAGNOSTICS=HOTEL.DIA
```

To change the ADA qualifiers executed by the LSE COMPILE command, you can use one of the following methods:

- Specify a dollar sign ($) as the first parameter to the LSE COMPILE command, appending the qualifiers you wish to change. For example:

  ```
  LSE> COMPILE $/NOOPTIMIZE
  ```

  This method is most useful when you want to change qualifiers on a one-time or infrequent basis.

- Modify the meaning of the LSE COMPILE command, using the LSE MODIFY LANGUAGE/COMPILE_COMMAND command. For example:

  ```
  LSE> MODIFY LANGUAGE/COMPILE_COMMAND="ADA/ANALYSIS_DATA"
  ```

  This method is useful if you want to use a particular qualifier change many times from the same editing buffer. If you then save your environment file after entering the MODIFY LANGUAGE command, and make the file part of the context for all of your editing sessions, the change will be permanent (until you change it again).

- Redefine the symbol ADA at the DCL level. For example:

  ```
  $ ADA == "ADA/ANALYSIS_DATA"
  ```

  This method is useful if you want to change qualifiers on a permanent or semipermanent basis. If you put the symbol definition in your LOGIN.COM file, it is defined permanently (until you change it again). The symbol is passed to your subprocess each time the LSE COMPILE command is executed.

When the LSE COMPILE command finishes executing, you can review any errors by entering the LSE REVIEW command. Alternatively, you can go directly from compile to review mode by entering the LSE COMPILE/REVIEW command.

The REVIEW and COMPILE/REVIEW commands display any diagnostic messages that result from a compilation. LSE displays the compilation errors in one window and the corresponding source code in a second window so that you can review your errors while examining the associated source code.

You can use the LSE REVIEW command to review the diagnostics and source code for a number of units at the same time by concatenating the diagnostics files for the units and then using the REVIEW/FILE command. For example, you can use the ADA/DIAGNOSTICS (or ACS COMPILE/DIAGNOSTICS) command outside of the editor to compile a number of Ada source files:

```
$ ADA/DIAGNOSTICS A
$ ADA/DIAGNOSTICS B
$ ADA/DIAGNOSTICS C
    .
    .
    .
```

Then, you can concatenate the resulting .DIA files using the DCL COPY command:

```
$ COPY *.DIA ALL_ERRORS.DIA
```

Then, you can invoke LSE, and use the following REVIEW command to review the errors—and use CTRL/G (GOTO SOURCE) to have the editor bring up the corresponding source file for each error:

```
LSE> REVIEW/FILE=ALL_ERRORS
```

NOTE

When you use the /DIAGNOSTICS qualifier with the ACS COMPILE command, it applies only to sources files that are compiled again. It has no effect on units that are recompiled (no .DIA files are produced). Similarly, you can use the /DIAGNOSTICS qualifier with the ACS RECOMPILE command, but it has no effect.

Table C–2 summarizes the LSE commands for compiling your program and reviewing any errors.

Table C–2: VAX LSE Commands for Compiling a Program and Reviewing Errors

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer. You can enter this command with the /REVIEW qualifier to put LSE in REVIEW mode immediately after the compilation.
REVIEW	None	Puts LSE into REVIEW mode and displays any errors resulting from the last compilation.
END REVIEW	None	Removes the buffer $REVIEW from the screen; returns the cursor to a single window containing the source buffer.
GOTO SOURCE	CTRL/G	Moves the cursor to the source buffer that contains the error.
NEXT STEP	CTRL/F	Moves the cursor to the next error in the buffer $REVIEW.
PREVIOUS STEP	CTRL/B	Moves the cursor to the previous error in the buffer $REVIEW.
None	{ Down arrow / Up arrow }	Moves the cursor within a buffer.

C.1.5 Sample LSE Session

This section shows parts of an LSE session used to develop the Ada source program in Example C–1. The program calls the VMS system routine SYS$ASCTIM, which returns the current time as an ASCII string. Instructions and explanations precede each step (or group of steps) in the development of the program.

The program development steps show expansions of placeholders and tokens to produce the following program elements:

- A context clause
- An Ada main program
- An object declaration
- A call to a VMS system routine in package STARLET

- An input-output operation in package TEXT_IO
- An **if** statement

Control keys and commands that manipulate tokens and placeholders are mentioned as appropriate; see Section C.1.3 for a more complete list of control keys and commands.

Remember that braces ({}) enclose required placeholders; brackets ([]) enclose optional placeholders. When you erase an optional placeholder using the ERASE PLACEHOLDER/FORWARD command or CTRL/K, LSE also deletes any associated text before and after that placeholder.

NOTE

Keywords such as **procedure**, **type**, **exception**, and so on can be tokens as well as placeholders; thus, any time you are in the VAX Ada language environment, you can type one of these words and press CTRL/E to expand the construct.

Example C–1: Complete Ada Program Developed Using LSE

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if CONDITION_HANDLING.SUCCESS(RET_STATUS) then
        TEXT_IO.PUT_LINE (ITEM => CURRENT_TIME);
    else
        TEXT_IO.PUT_LINE("Call to ASCTIM failed.");
    end if;
end LSE_EXAMPLE;
```

Step 1: Creating a Main Program with a Context Clause

When you use LSE to create a new VAX Ada program, the initial string, {compilation_unit}, appears at the top of the screen. For example:

```
{compilation_unit}
[End of file]
```

(The [End of file] indicator appears at all times when you are in an LSE buffer; it is shown here only for completeness and does not appear in other examples in this section.)

Using CTRL/E, expand the initial string to produce a menu, and press the down arrow key until the cursor points at the option {procedure body}. Press the RETURN key. The following appears on your screen:

```
-- [source_file_header_comment]
[context_clause]
{procedure_body}
```

Form a context clause by first pressing CTRL/K to erase the [source_file_ header_comment] placeholder, and then pressing CTRL/E to expand the [context_clause] placeholder. The following appears on your screen:

```
[with_clause]...
[use_clause]...
[context_clause]...
{procedure_body}
```

Press CTRL/E to expand the [with_clause] placeholder, with the following result:

```
with {unit_identifier}...;
[with clause]...
[use_clause]...
[context_clause]...
{procedure_body}
```

Type TEXT_IO in place of the {unit_identifier} placeholder, and press CTRL/N to get to the next {unit_identifier} placeholder, which LSE automatically creates each time you add a unit identifier to the context clause:

```
with TEXT_IO, {unit_identifier}...;
[with_clause]...
[use_clause]...
[context_clause]...
{procedure_body}
```

Type STARLET in place of the {unit_identifier} placeholder. Repeat this process to add CONDITION_HANDLING to the context clause, then press CTRL/K four times to delete the remaining {unit_identifier} placeholder and the [with_clause], [use_clause], and [context_clause] placeholders. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
{procedure_body}
```

Form the main procedure by pressing CTRL/E to expand the {procedure_ body} placeholder. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure {procedure_identifier} [formal_part] is
-- [procedure_header_comment]
     [declarative_part]
begin
     {statement}...
[exception_part]
end [procedure_identifier];
```

Type LSE_EXAMPLE in place of {procedure_identifier}; note that LSE automatically replaces the [procedure_identifier] placeholder at the end statement for you. Now, erase the [formal_part] placeholder by pressing CTRL/K. Press CTRL/K again to erase the [procedure_header_comment] placeholder. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    [declarative_part]
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Now that the main procedure template is formed, you can expand the declarative and statement parts as described in steps 2, 3, and 4.

Step 2: Declaring Objects

Using the main procedure template at the end of step 1, press CTRL/E to expand the [declarative_part] placeholder. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    [basic_declarative_item]...
    [later_declarative_item]...
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Press CTRL/E to expand the [basic_declarative_item] placeholder. This placeholder is a menu placeholder; choose {basic_declaration} from the menu. The following appears on your screen, with a menu of basic declarations:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    {basic_declaration}
    [basic_declarative_item]...
    [later_declarative_item]...
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Choose {object_declaration} from the menu. LSE automatically presents you with another menu of possible object declarations; choose the object appropriate for declaring a STRING variable. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    {identifier}...:[constant]{subtype_indication}:=[initial_value];
    [basic_declarative_item]...
    [later_declarative_item]...
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Type CURRENT_TIME in place of the {identifier} placeholder. Press CTRL/K twice: once to erase the rest of the {identifier} placeholder, and once to erase the [constant] placeholder. Press CTRL/E to expand the {subtype_indication} placeholder. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME: {type_mark} [constraint] := [initial_value];
    [basic_declarative_item]...
    [later_declarative_item]...
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Type STRING in place of the {type_mark} placeholder. Press CTRL/N to get to the [constraint] placeholder, and type (1..23) to replace the placeholder. Press CTRL/N to give the object CURRENT_TIME an initial value by replacing the [initial_value] placeholder with the aggregate (others => ' '). The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    [basic_declarative_item]...
    [later_declarative_item]...
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Press CTRL/N to get to the [basic_declarative_item] placeholder. Expand the placeholder, and create another object, RET_STATUS, which should be of type CONDITION_HANDLING.COND_VALUE_TYPE. Press CTRL/K twice to erase the the extra [basic_declarative_item] and [later_declarative_item] placeholders. The following appears on the screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    {statement}...
[exception_part]
end LSE_EXAMPLE;
```

Now, you can expand the {statement} part of the template as described in steps 3 and 4.

Step 3: Using LSE to Write a Package STARLET System Service Call

Using the template at the end of step 2, type STARLET.ASCTIM in place of the {statement} placeholder, as follows. (Alternatively, you can type STARLET and expand it into a menu of the routines available from the Ada predefined package STARLET.)

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Press CTRL/E immediately after typing STARLET.ASCTIM to expand the token. The following appears on the screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => {status},
        [TIMLEN => {timlen}],
        TIMBUF => {timbuf},
        [TIMADR => {timadr}],
        [CVTFLG => {cvtflg}]);
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Type RET_STATUS in place of the {status} placeholder, and press CTRL/K to erase the next (optional) parameter. Type CURRENT_TIME in place of the {timbuf} placeholder, and use CTRL/K to erase the remaining optional parameters. The following appears on the screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Step 4: Writing an if Statement and Calling a TEXT_IO Procedure

Using the template from the end of step 3, press CTRL/E to expand the [statement] placeholder, and choose the **if** statement from the menu. The following appears on the screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if {condition} then
        {statement}...
    [elsif_part]...
    [else_part]
    end if;
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Type the boolean function CONDITION_HANDLING.SUCCESS(RET_STATUS) in place of the {condition} placeholder, with the following result:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if CONDITION_HANDLING.SUCCESS(RET_STATUS) then
        {statement}...
    [elsif_part]...
    [else_part]
    end if;
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Press CTRL/N to get to the {statement} placeholder. Type TEXT_IO.PUT_LINE in place of the {statement} placeholder and press CTRL/E to obtain the syntax for PUT_LINE from LSE (all of the Ada input-output operations are also LSE tokens). The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if CONDITION_HANDLING.SUCCESS(RET_STATUS) then
        TEXT_IO.PUT_LINE ([FILE => {file_identifier}],
                          ITEM => {string_or_char_expression});
        [statement]...
    [elsif_part]...
    [else_part]
    end if;
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Press CTRL/K to delete the first (optional) parameter, and type CURRENT_TIME in place of the {string_or_char_expression} placeholder. Press CTRL/K twice more to erase the [statement] and [elsif_part] placeholders. The following appears on your screen:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if CONDITION_HANDLING.SUCCESS(RET_STATUS) then
        TEXT_IO.PUT_LINE (ITEM => CURRENT_TIME);
    [else_part]
    end if;
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Press CTRL/E to expand the [else_part] placeholder, and substitute another TEXT_IO.PUT_LINE statement for the resulting {statement} placeholder, with the following result:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if CONDITION_HANDLING.SUCCESS(RET_STATUS) then
        TEXT_IO.PUT_LINE ( ITEM => CURRENT_TIME);
    else
        TEXT_IO.PUT_LINE( ITEM => "Call to ASCTIM failed.");
        [statement]...
    end if;
    [statement]...
[exception_part]
end LSE_EXAMPLE;
```

Use CTRL/K to erase the [statement] and [exception_part] placeholders. The complete program now appears on the screen, as follows, and can be compiled:

```
with TEXT_IO, STARLET, CONDITION_HANDLING;
procedure LSE_EXAMPLE is
    CURRENT_TIME : STRING (1..23) := (others => ' ');
    RET_STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
    STARLET.ASCTIM (
        STATUS => RET_STATUS,
        TIMBUF => CURRENT_TIME);
    if CONDITION_HANDLING.SUCCESS(RET_STATUS) then
        TEXT_IO.PUT_LINE ( ITEM => CURRENT_TIME);
    else
        TEXT_IO.PUT_LINE( ITEM => "Call to ASCTIM failed.");
    end if;
end LSE_EXAMPLE;
```

C.2 Using VAX Ada with SCA

SCA is an interactive tool for cross-referencing and statically analyzing source code. You can use it with most VAX languages, including VAX Ada. SCA stores data generated by the VAX Ada compiler in an SCA library. The data in an SCA library contains information about all the symbols, modules, and files encountered during compilation of your source files.

Cross-referencing provides information about program symbols and source files. SCA provides the following cross-referencing features:

- You can locate symbols and occurrences (uses) of those symbols in your source code. SCA allows you to obtain information on one or several

symbols; you can also obtain information on partial symbols by using wildcard characters.

- You can find symbols or files of particular kinds (such as subprogram names, operators, variable names, or source files).
- You can find specific occurrences (or uses) of the symbols you are interested in. For example, you can look for symbol declarations, exceptions, calls to subprograms or operators, and so on.

Static analysis provides information on how subprograms, symbols, and files are related. SCA provides the following static analysis features:

- You can display calls to a particular subprogram, and then go to the source location of those calls (SCA FIND/REFERENCE=CALL).
- You can display call tree information related to a particular subprogram; you can also display calls to or from a particular subprogram (SCA VIEW CALL_TREE).

Note that because Ada is strongly typed and subprogram calls are checked by the compiler, the SCA CHECK CALLS command has no effect for VAX Ada.

SCA is fully integrated with LSE to provide additional features. By using SCA with LSE, you can view any portion of an entire system and edit the related source files.

See Section C.1 for general and Ada-specific information on LSE; see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer* for detailed information on both LSE and SCA.

C.2.1 Setting Up an SCA Environment

To use SCA to analyze VAX Ada source code, you must take the following steps:

1. Create an SCA library.
2. Use the VAX Ada compiler to generate data analysis files for each compilation unit you want to analyze.
3. Load the information from the data analysis files into your SCA library.

The following sections describe these steps in more detail.

C.2.1.1 Creating an SCA Library

To use SCA, you must have an SCA library in which to store the analysis data that the VAX Ada compiler collects. To create an SCA library, first create a VMS directory. For example:

```
$ CREATE/DIRECTORY [.MY_SCA_LIBRARY]
```

Next, initialize and set the library with the SCA CREATE LIBRARY command. For example:

```
$ SCA CREATE LIBRARY [.MY_SCA_LIBRARY]
```

If you have an existing SCA library that has been initialized, you make its contents visible to SCA by setting it with the SCA SET LIBRARY command. For example:

```
$ SCA SET LIBRARY [.EXISTING_SCA_LIBRARY]
```

VAX Ada provides a predefined SCA library for the Ada predefined units. The name of this library is ADA$SCA_PREDEFINED. To make its contents visible to SCA, add it to your SCA library list using the SCA SET LIBRARY command. For example:

```
$ SCA SET LIBRARY [.MY_SCA_LIBRARY],ADA$SCA_PREDEFINED
```

Note that ADA$SCA_PREDEFINED is a system library, and thus is generally read-only (for example, you cannot use the SCA DELETE MODULE, REORGANIZE, or VERIFY commands with this library).

C.2.1.2 Generating Data Analysis Files

The ADA, ACS COMPILE, and ACS RECOMPILE commands all have an optional /ANALYSIS_DATA[=filespec] qualifier that causes data analysis files to be output for each file that is compiled. By default, the data analysis files are created in your current default directory. By default, they have the same names as the names of the files that were compiled, and they have a file type of .ANA. For example:

```
$ ADA/LIST/ANALYSIS_DATA ADA_PACKAGE,ADA_PROGRAM
```

This command compiles the input files ADA_PACKAGE.ADA and ADA_PROGRAM.ADA into the current Ada program library, and generates two output files for each input file in the current default directory: ADA_PACKAGE.LIS, ADA_PACKAGE.ANA, ADA_PROGRAM.LIS, and ADA_PROGRAM.ANA.

Note that unlike the ADA command, the ACS RECOMPILE and COMPILE commands operate on compilation units, not source files, and different sets of data analysis files can result based on the way these commands operate. The ACS RECOMPILE command creates a .ANA file from the copied source file of each unit involved in the recompilation. The ACS COMPILE command creates a .ANA file for each original source file that it must compile again and a .ANA file for each unit that it recompiles. In both cases, if no other directory is specified in the /ANALYSIS_DATA[=filespec] command, the .ANA files appear in the current default directory.

C.2.1.3 Loading Data Analysis Files into a Local Library

Once you have an SCA library and have generated data analysis files for your source code during compilation, you must load the information in the data analysis files into your SCA library. For example:

```
$ SCA LOAD ADA_PACKAGE,ADA_PROGRAM
```

This command loads your current library with the information contained in ADA_PACKAGE.ANA and ADA_PROGRAM.ANA.

C.2.2 Using SCA for Cross-Referencing

Once you have set up your SCA environment, you can ask for symbol or file information by using the SCA command FIND. The FIND command has the following form:

```
FIND [/qualifier...] [name-expression[,...]]
```

The possible qualifiers are /FILE and /SYMBOL_CLASS; /SYMBOL_CLASS is the default. The name can represent any of the following entities:

Name	A series of characters that uniquely identifies a symbol or a file
Item	An appearance of a symbol (such as a variable, constant, label, or procedure) or a file
Occurrence	The use of a symbol or a file

The name expression can be explicit or can contain wildcards. For example:

```
$ SCA FIND ABC,XY%
```

SCA is integrated with LSE. Thus, you can execute any SCA command from within LSE. Once you are inside the editor, you press CTRL/Z to get the LSE> prompt; you can type any SCA command at the prompt, and press the RETURN key to execute the command. For example:

```
LSE> FIND ABC,XY%
```

When you first enter a FIND command within LSE, you initiate a query session. Within this context, the integration of LSE and SCA provides commands that can be used only within LSE. Table C–3 lists these commands.

Table C–3: VAX LSE Commands for Making SCA Queries

Command	Function
{ NEXT / PREVIOUS } { NAME / ITEM / OCCURRENCE / QUERY / STEP }	Allows you to step through one or more query buffer displays within LSE
GOTO SOURCE	Displays the source corresponding to the current query item
GOTO DECLARATION	Positions the cursor on a symbol declaration in one window, and displays the source code that contains the symbol declaration in another window

The following sections discuss the use of the SCA FIND command and the LSE-related SCA commands in more detail.

C.2.2.1 Finding Files

The SCA FIND/FILE command is designed to help you find information on the files involved in your program. Since Ada programs are structured as sets of compilation units, rather than files, this command has little use with VAX Ada programs except to tell you which compilation units (called modules in SCA) are in which source files.

C.2.2.2 Finding Ada Symbols

The SCA FIND/SYMBOL_CLASS command is designed to help you find information on symbols in your program. This command has a number of qualifiers for helping you find the information about a symbol or group of symbols. Those qualifiers that have Ada-specific features or interpretations are explained in the following sections.

Note that some Ada constructs fall into more than one of the categories defined by the various qualifiers. For example, a generic instantiation is both a declaration and a reference. If, for example, a generic package is instantiated, a declaration is generated for the instantiation; a reference is generated for the generic package. Thus, the following instantiation will

generate both a declaration of package MY_INT_SORT and a reference to generic package SORT (as well as a reference to type INTEGER):

```
package MY_INT_SORT is new SORT(INTEGER);
```

Also note that SCA recognizes Ada operators as symbols; you can use any of the SCA FIND/SYMBOL_CLASS qualifiers to query SCA about the use of operators in your Ada source code. When querying SCA about an operator using the FIND/SYMBOL_CLASS command, you must enclose the operator in quotation marks. For example:

```
LSE> FIND/SYMBOL_CLASS/DECLARATIONS "+"
```

To query SCA about a multiplication operator (*), you must go to the source code and use the SCA FIND/INDICATED command (see Section C.2.3). SCA recognizes this operator as a wildcard character even when it is inside quotation marks. The FIND/INDICATED command can be used with any of the FIND/SYMBOL_CLASS qualifiers.

C.2.2.2.1 Declarations

The /DECLARATIONS[=(option[,...])] qualifier allows you to query SCA for information on symbol declarations. The option keywords are PRIMARY, ASSOCIATED, EXPLICIT, IMPLICIT, VISIBLE, HIDDEN, ALL, and NONE. Note the following points about the use of this qualifier and its keywords with VAX Ada source code:

- In a very general sense, Ada specifications (package, subprogram, and task) are considered to be associated declarations; the corresponding bodies are considered to be primary declarations. Thus, a main subprogram (which has no separate specification) has only a primary declaration.

  SCA recognizes the pairs of associated and primary declarations in Ada source code listed in Table C–4.

  For Ada constructs that have both a primary and an associated declaration, you can use the LSE GOTO DECLARATION/INDICATED/CONTEXT_DEPENDENT command (by default bound to the PF1-CTRL/D keypad key) to toggle between the two declarations in the source code. See Section C.2.3 for more information on commands that involve the /INDICATED qualifier.

- The EXPLICIT option keyword provides information on explicit declarations in the source code. Declarations that result from a generic instantiation are also considered to be explicit (even though they are hidden).

- The IMPLICIT option keyword provides information on implicit operators derived from type declarations (for example, the operations associated with type INTEGER are considered to be declared implicitly).
- The VISIBLE option keyword provides information on declarations that appear only in the source code (declarations that result from a generic instantiation are not visible).
- The HIDDEN option keyword provides information on declarations that are presumed, but not declared, in the source code. Implicit declarations (such as operations associated with a type) and declarations that result from a generic instantiation are hidden declarations.
- The ALL and NONE option keywords provide either all or none of the information available on symbol declarations.

Table C–4: Ada Constructs Associated with SCA PRIMARY and ASSOCIATED Keywords

Ada Constructs Associated with the PRIMARY Keyword	Ada Constructs Associated with the ASSOCIATED Keyword
accept statement	entry [family] declaration
full constant declaration	deferred constant declaration
[generic] package body	[generic] package specification
[generic] subprogram body	[generic] subprogram specification
task [type] body	task [type] specification
full type declaration	incomplete or private type declaration
accept formal parameters in a task [type] body	entry formal parameters in a corresponding task [type] specification
subprogram formal parameters in a body	subprogram formal parameters in a corresponding specification
discriminants in a full type declaration	corresponding discriminants in an incomplete or private type declaration

C.2.2.2.2 References

The /REFERENCES[=(option[,...])] qualifier allows you to query SCA information on symbol references. The option keywords are READ (or FETCH), WRITE (or STORE), ADDRESS (or POINTER), CALL, OTHER, VISIBLE, HIDDEN, ALL, and NONE. Note the following points about the use of this qualifier and its keywords with VAX Ada source code:

- The READ and WRITE keywords provide information on symbol values that have been read or written.

- The ADDRESS keyword has no meaning for VAX Ada.
- The CALL keyword provides information on calls to subprograms (note that operators are also subprograms).
- The OTHER keyword provides information on all references that are not READ, WRITE, or CALL references. For example, references to exceptions in a raise statement are other references, as are references to exceptions in an exception handler.
- The VISIBLE keyword provides information on all references that are in the current source code.
- The HIDDEN keyword provides information on all references that are related to the current source code, but are not directly visible (for example, the operations in an instantiation of package DIRECT_IO).
- The ALL and NONE option keywords provide either all or none of the information available on symbol references.

C.2.2.2.3 Symbol Classes

The /SYMBOL_CLASS[=(symbol_class[,...])] qualifier allows you to select various classes of symbols. The following keywords are provided (synonyms that are less relevant to Ada are in parentheses):

ADDRESS (or POINTER)
ARGUMENT
COMPONENT (or FIELD)
CONSTANT (or LITERAL)
EXCEPTION
FILE
LABEL
GENERIC
MACRO
PACKAGE (or MODULE, PROGRAM)
PSECT
FUNCTION or PROCEDURE (or ROUTINE, SUBROUTINE)
TASK
TYPE
UNBOUND
VARIABLE
OTHER
ALL
NONE

When using the SCA FIND/SYMBOL/SYMBOL_CLASS command for symbols that are not alphanumeric identifiers (such as subtraction, concatenation, and addition operators), you must enclose the symbols in quotation marks; for example: SCA FIND "-", "&", "+".

Note the following points about the use of this qualifier and its keywords with VAX Ada source code:

- The ADDRESS keyword has no meaning for VAX Ada.
- The ARGUMENT keyword lists subprogram formal parameters.
- The COMPONENT keyword lists record components and discriminants.
- The CONSTANT keyword lists constants.
- The EXCEPTION keyword lists exceptions.
- The FILE keyword has no meaning for VAX Ada.
- The LABEL keyword lists labels and loop identifiers.
- The GENERIC keyword lists generic packages and subprograms.
- The MACRO keyword has no meaning for VAX Ada.
- The PACKAGE keyword lists packages.
- The PSECT keyword has no meaning for VAX Ada.
- The FUNCTION or PROCEDURE keyword lists all subprograms (including operators), entries, and accept statements (all are listed as a result of either keyword).
- The TASK keyword lists all task objects.
- The TYPE keyword lists all types.
- The UNBOUND keyword lists all Ada attributes and pragmas. Pragmas that are compilation units—MEMORY_SIZE, SYSTEM_NAME, and LONG_FLOAT—do not appear in SCA displays.
- The VARIABLE keyword lists all Ada objects.
- The OTHER keyword has no meaning for VAX Ada (all symbol classes are accounted for with symbol class keywords).
- The ALL and NONE option keywords provide either all or none of the information available on symbol classes.

Note that single tasks appear as having a task type specification declaration, a task type body declaration, and a task object declaration. Task types appear as having a task type specification declaration and task body declaration; task objects appear as having only a task object declaration.

C.2.3 Navigating Through Ada Source Code

The best way to navigate through code is to use SCA from inside VAX LSE. The following navigation features are especially useful in analyzing Ada source code:

NOTE

When you use these commands with an operator, you must first enclose the operator in a select range (use the SELECT command bound to the period (.) key on the keypad and the CHAR command bound to the three (3) key on the keypad).

If you do not use the command with an operator, and the operator is enclosed in quotation marks in the source code, you should include only the operator symbol (and not the quotation marks) in the select range.

- FIND/INDICATED—You must position the cursor on a symbol in the source code before you can execute this command; you can use this command to find context-dependent occurrences of a particular symbol. You can use any of the FIND/SYMBOL_CLASS qualifiers with this command to control the kind of information displayed for the indicated symbol. However, the /REFERENCES[=(option[,...])] and /DECLARATIONS[=(option,...)] qualifiers are perhaps the most useful.
- GOTO DECLARATION/INDICATED/CONTEXT_DEPENDENT (PF1-CTRL/D)—You can use this command to toggle between primary and associated declarations of a particular symbol. For example, if the cursor is on a primary declaration, this command will cause the cursor to be positioned at the associated declaration, and vice versa.

  You can also use this command on symbol references. In such cases, the cursor is positioned on the location that determines the meaning of the symbol in the context in which the command was executed. For example, if you are working with an Ada private type, and you position the cursor on a reference to the type in the package body, the cursor is moved to the complete declaration of the type in the package body. If you position the cursor on a reference to the type that is outside of the package, the cursor is moved to the declaration of the type in the package specification.
- GOTO DECLARATION/INDICATED/PRIMARY (CTRL/D)—You must position the cursor on a symbol in the source code before you can execute this command; you can use the CTRL/D key binding to quickly find the body of a subprogram or package, operator, accept statement, and so on (see Table C–4). If the symbol does not have a primary declaration

(as is the case for specifications that have no bodies), SCA will issue an informational message to that effect.

- GOTO DECLARATION/INDICATED/ASSOCIATED—You must position the cursor on a symbol in the source code before you can execute this command; you can use this command to find the specification of a subprogram or package, operator, entry declaration, and so on (see Table C–4). If the symbol does not have an associated declaration (as is the case for subprograms that have no separate specifications), SCA will issue an informational message to that effect.

  Note that although this command has no key bound to it, you can bind it to a key of your choice using the LSE DEFINE KEY command.

- GOTO SOURCE (CTRL/G)—You must position the cursor on an entity in an SCA query display. When you execute this command, the cursor is positioned at the entity's location in the actual source code. This command can be used on all Ada units, including the predefined units in ADA$SCA_PREDEFINED.

C.2.4 Using SCA for Static Analysis

The SCA CHECK CALLS and VIEW CALL_TREE commands allow you to statically analyze your source code (display calls to a particular subprogram and display call tree information). When using these commands with VAX Ada source code, note the following points:

- The SCA CHECK CALLS command has no effect for Ada because Ada is strongly typed; the compiler ensures that the actual parameters in subprogram calls correctly match the formal parameters in the corresponding subprogram specifications.
- The SCA VIEW CALL_TREE command stops when it encounters a call to a subprogram resulting from a generic instantiation.
- When the SCA VIEW CALL_TREE command is entered with the name of an overloaded subprogram, it will produce n distinct call trees for each of the n overloadings.

C.2.5 Multimodular Development

The cross-referencing and static analysis features of SCA are especially useful during the implementation and maintenance phases of a project that involves many programming modules. For example, the project team work area in Figure C–1 contains a set of source modules. (The team might use a code management tool, such as VAX DEC/CMS, to keep track of these modules in their various development stages.) When the team compiles the source code, SCA generates the source information it requires (that is, data analysis files with the file type .ANA); then the team loads this information into a previously established project SCA library.

When a team member wants to do additional development work on specific modules, that member sets up an individual work area, which might contain the following:

- Copies of source and object modules from the project libraries
- Local SCA libraries that contain copies of the module information

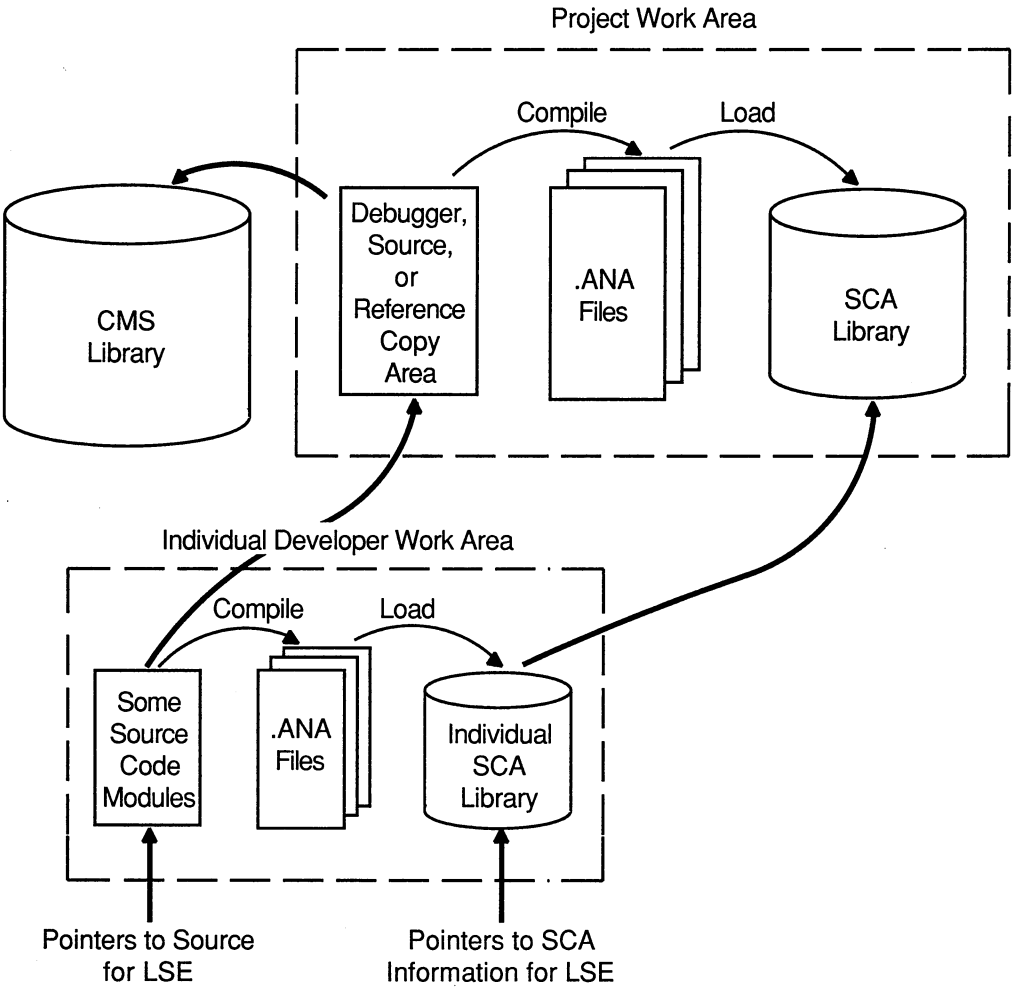
To make available all the capabilities of SCA/LSE integration, the team member informs LSE of the locations of that member's current sources, and related source information. Using LSE, all team members can effectively see through their own individual work areas to the project work area and possibly to other individual work areas.

The following sections provide a general overview of SCA and discuss some of the commands that are available to you when you use SCA within LSE. Information is also provided that is necessary for using SCA to analyze VAX Ada programs. For detailed information on SCA, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

C.2.6 Additional Ada-Specific SCA Considerations

Many of the Ada-specific SCA considerations have been presented in previous sections. The following sections note additonal considerations that can aid your use of SCA with VAX Ada source code.

Figure C–1: Using LSE and SCA for Multimodular Development



C.2.6.1 Library Differences

SCA libraries and VAX Ada program libraries have some differences, and the commands for creating, deleting, and setting SCA libraries behave somewhat differently from similar ACS commands. Table C–5 summarizes these differences.

Table C–5: Comparison of SCA and ACS Library Characteristics

SCA Libraries	VAX Ada Libraries
The SCA CREATE LIBRARY command requires that a VMS directory already exist before the command is entered.	ACS CREATE LIBRARY and CREATE SUBLIBRARY commands create a VMS directory if one does not already exist.
The SCA CREATE LIBRARY command also defines the library it creates to be the current, active SCA library.	The ACS CREATE LIBRARY and CREATE SUBLIBRARY commands do not automatically set the current program library.
To delete an SCA library and its contents, you must use the DCL DELETE command.	To delete a VAX Ada library, you must use the ACS DELETE LIBRARY or DELETE SUBLIBRARY command.
The SCA CREATE and SET LIBRARY commands allow you to create and set a group of libraries, which then together act as a single, virtual library.	The ACS CREATE LIBRARY, CREATE SUBLIBRARY, and SET LIBRARY commands operate on only one library at a time; you can have only one library as your current library at any one time.
You must manage your SCA libraries and library lists yourself. For example, you must update your SCA libraries when units are recompiled.	VAX Ada libraries are updated by the VAX Ada program library manager.
Data analysis files are produced and can be loaded into an SCA library, whether or not the compilation was successful.	The VAX Ada compiler updates the current program library only when a compilation is successful.
SCA cannot be used across DECnet.	VAX Ada libraries can be accessed across DECnet; see Chapter 5.

(continued on next page)

Table C–5 (Cont.): Comparison of SCA and ACS Library Characteristics

SCA Libraries	VAX Ada Libraries
If you load a unit specification (a package, function, procedure, generic package, generic function, or generic procedure specification) into an SCA library, and then you load a unit body that has a matching name but that declares a different kind of unit, SCA will replace the existing specification with the new body. For example, if you load a procedure specification into your SCA library and then load a function body with the same name, SCA will delete the procedure specification and replace it with the function body.	The VAX Ada program library manager does not allow you to replace a specification with a body.

C.2.6.2 Ada-Related Effects and Restrictions

Note the following Ada-related effects and restrictions in SCA queries:

- The SCA SHOW MODULE/BRIEF command (the default when wildcards are used in or for module names) may often list a given module twice. This is a feature for VAX Ada units that have both a specification and a body: one module represents the specification, and the other represents the body. The SCA SHOW MODULE/FULL command also lists modules twice, but provides enough information to allow you to distinguish specifications and bodies.
- Due to an SCA restriction, the full name of an Ada library unit cannot exceed 1024 characters in query displays (unit names greater than 1024 characters are truncated from the right in the displays). This restriction generally applies only to deeply nested subunits. This restriction also applies only to displays; internally, SCA recognizes names of any length.

Appendix D

Program Library and Sublibrary Structure and Contents

A VAX Ada program library or sublibrary is a dedicated VMS directory that is recognized by the VAX Ada compiler and program library manager.

Before any compilation has occurred, a program library contains only two files, which were created when the program library was created:

- The *library index file* (ADALIB.ALB)
- A file used for program library version control (ADA$LIB.DAT)

The library index file distinguishes a program library or sublibrary from other VMS directories, and is used by the program library manager for information like the following:

- To associate compilation unit and subunit names with their associated VMS file specifications
- To record the date and time when the VMS files were created or revised
- To maintain references to entered units
- To maintain a reference to the parent library if the library is a sublibrary

The program library manager and the compiler use and update this data to keep track of successfully compiled units and their order of compilation. Whenever a previously compiled unit is successfully compiled again, earlier index entries are revised to reflect the new file specifications and compilation date and time.

The current program library (see Section 2.1.2) is updated each time one of the following VAX Ada commands is executed successfully:

```
DCL ADA
ACS COMPILE
ACS RECOMPILE
ACS COPY
ACS ENTER
```

In the course of program development, a number of files are created or deleted from the program library or sublibrary, and data may be read, written, or deleted from these files and the library index file, depending on the operation performed (compilation, linking, program library management, and so on).

Each time a unit is successfully compiled into the current program library, the following files are created or accounted for (their VMS file types appear in parentheses):

- An *object file* (.OBJ) is usually created for each compilation unit. The object file contains the machine code instructions for that unit. Note that the compiler may not always create an object file for each compilation unit. For example, compiling a library function specification does not create an object file.
- A *compilation unit file* (.ACU) is created for each compilation unit. The file contains data that is used to support separate compilation, linking, and program library management. The data includes the name of the compilation unit; whether it is a specification, a body, or a subunit; use of certain pragmas, and so on. Also, the file identifies all library specifications that the given compilation unit depends on.
- Unless suppressed by the /NOCOPY_SOURCE compilation qualifier, a *copied source file* (.ADC) is created for each compilation unit. This file contains a copy of the Ada source text for that compilation unit. It is used by the ACS RECOMPILE command for recompiling obsolete units (see Chapter 3). It is also used by the VMS Debugger to display source lines during debugging (see Chapter 6).
- Unless suppressed by the /NONOTE_SOURCE compilation qualifier, the file specification of the source file (.ADA) is noted in the library index file. This source file specification is used by the ACS COMPILE command for compiling units from external source files (see Chapter 3). If the /NOCOPY_SOURCE qualifier is in effect, then this source file is used by the VMS Debugger to display source lines during debugging.

Other files created during compilation and linking (compiler listing files, linker map files, and so on) are created in your current default directory or in any other directory you may have specified for them with the appropriate compilation or linking commands.

When assigning file specifications to program library files, the program library manager and the compiler use the file-name conventions for source files described in Chapter 1.

Whenever a unit is copied into the current program library, its associated files are copied. Whenever a previously compiled unit is successfully compiled again or copied and replaced, earlier versions of the associated files are deleted.

One set of object, compilation unit, and copied source files is created for each compilation unit, not for each file submitted to the compiler. The contents of the program library would be the same if the source code for these compilation units had been arranged in one, two, or three source files. If the compiler detects a fatal or user error during the compilation of a given compilation unit, the program library is not updated for that unit. However, if the compiler issues only warning or informational messages, the program library is updated.

If several compilation units are submitted in one compilation and errors are detected, not all compilation units will have errors. In that case, the program library is updated only for those compilation units that do not have errors. See Chapter 3 for more information on compilation errors.

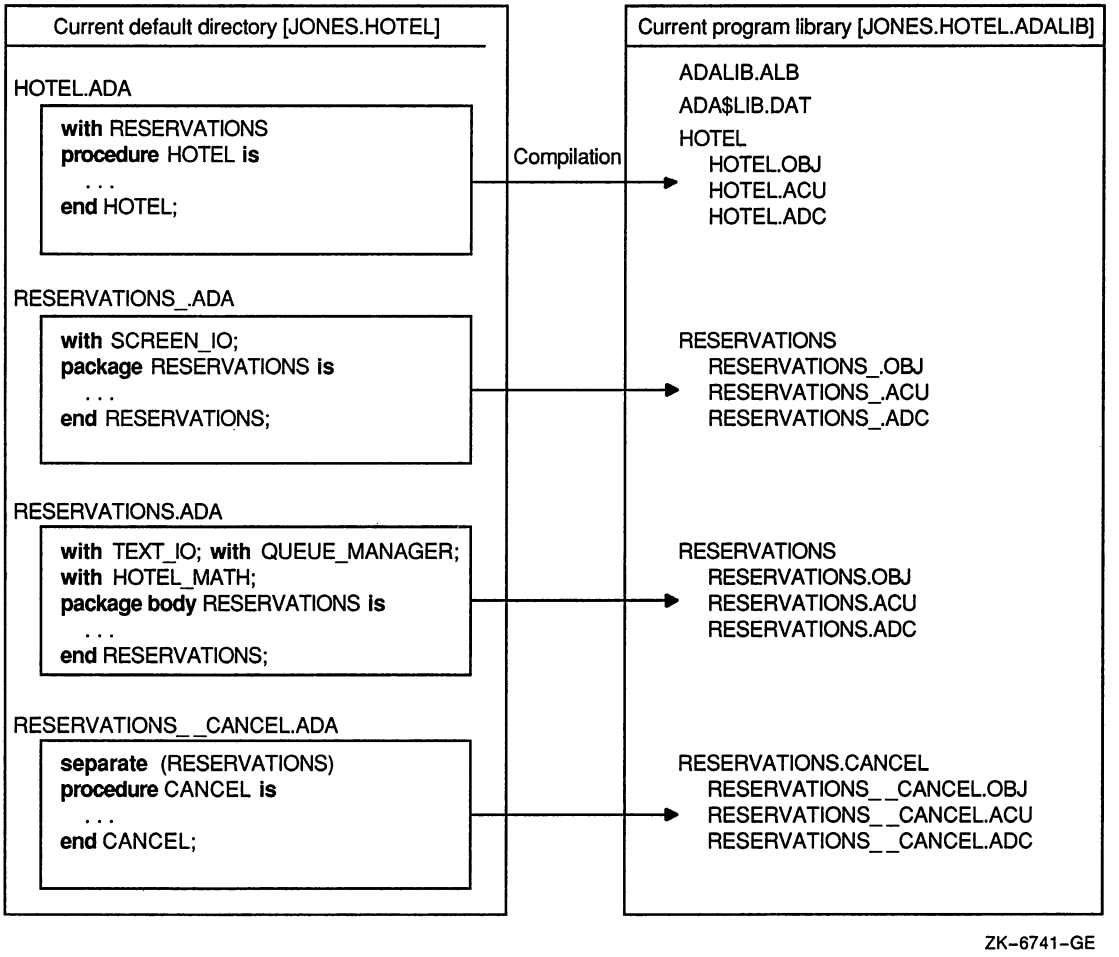
Figure D–1 shows a program library that contains the following set of compiled units:

- The procedure body HOTEL
- The specification and body for the package RESERVATIONS
- The subunit RESERVATIONS.CANCEL

Figure D–1 shows the relationship between the source files for these units and the compiled units and their associated files. The source files are in the current default directory [JONES.HOTEL], and the compiled units and their associated files are in the current program library [JONES.HOTEL.ADALIB].

The arrows in Figure D–2 show, in simplified fashion, how the library index file references units and associated files.

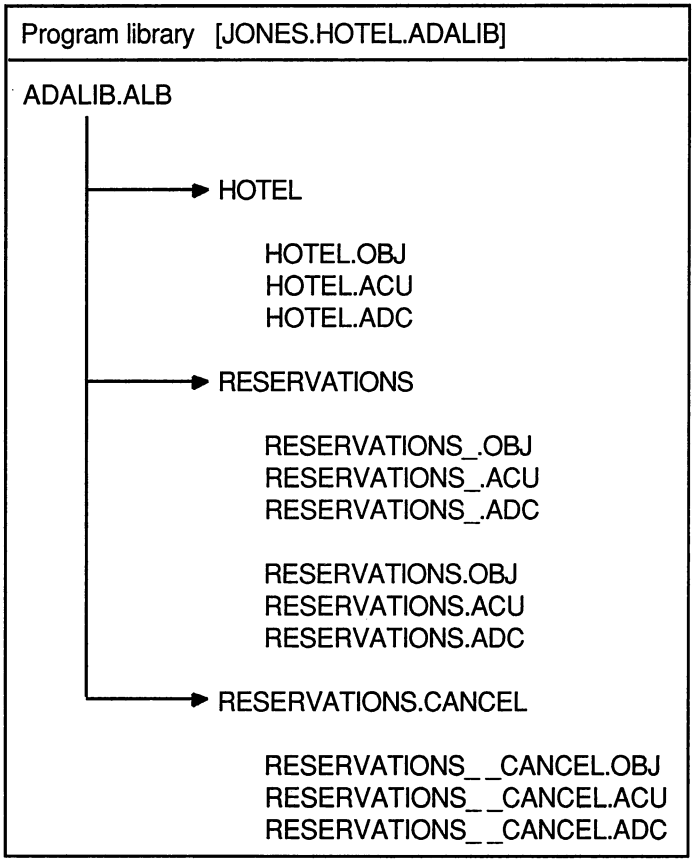
Figure D–1: Current Default Directory and Current Program Library After Compilation



You can use the /FULL qualifier with the ACS DIRECTORY command to display the following information about a unit or subunit in the current program library:

- The file specifications of the object (.OBJ), compilation unit (.ACU), and copied source (.ADC) files.

Figure D–2: Compilation Units as Entries in the Library Index File



ZK–6745–GE

- The file specification of the source file (.ADA). The file specification indicates the directory where the source file exists, and is preceded by the at character (@) to indicate that the file is not in the current program library. This information is used by the ACS COMPILE command.

For example:

```
$ ACS DIRECTORY/FULL SCREEN_IO
SCREEN_IO
    package specification           16-Apr-1989 13:36
        SCREEN_IO_.ACU;3
        SCREEN_IO_.OBJ;3
        SCREEN_IO_.ADC;3
      @ USER:[JONES.HOTEL]SCREEN_IO_.ADA;6
    package body                    16-Apr-1989 15:42
        SCREEN_IO.ACU;3
        SCREEN_IO.OBJ;3
        SCREEN_IO.ADC;2
      @ USER:[JONES.HOTEL]SCREEN_IO.ADA;5

Total of 2 units.
```

You can also use the /FULL qualifier with the ACS SHOW PROGRAM command to add to the display the names of the object file, copied source file, and .ACU file for each unit in the closure of a set of units. For example:

```
$ ACS SHOW PROGRAM/FULL SCREEN_IO

   .
   .
   .
SCREEN_IO
   Package specification
      Compiled:      16-Apr-1989 13:36
      Source file:    1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO_.ADA;1
      Object file:   SCREEN_IO_.OBJ;3
      Copied file:   SCREEN_IO_.ADC;3
      ACU file:      SCREEN_IO_.ACU;3

   Package body
      Compiled:      16-Apr-1989 15:42
      Source file:    1-Sep-1988 10:39 USER:[PROJ]SCREEN_IO.ADA;1
      Object file:   SCREEN_IO.OBJ;3
      Copied file:   SCREEN_IO.ADC;2
      ACU file:      SCREEN_IO.ACU;3
      With list:     TEXT_IO
   .
   .
   .
```

Appendix E

Efficient Compilation

This appendix presents information on memory and resource requirements for efficient compilation. It duplicates similar information in the *VAX Ada Installation Guide* and online release notes.

E.1 Memory Usage

Working set size and virtual address space are important factors in achieving efficient compilation of VAX Ada programs. The following sections analyze the results of sample compilations and suggest general guidelines for adjusting working set size and selecting an appropriate virtual address space.

E.1.1 Working Sets

To measure the effects of working set size on page faults and compilation rates, selected VAX Ada programs representing typical usage were compiled with a variety of working set sizes. The test compilations were run on a standalone system with the following configuration: a 13-megabyte VAXstation II with an RD53 system disk drive and an RD53 user disk drive running VMS Version 5.1.

Table E–1 gives brief descriptions of the source files compiled in the tests (each file represents one or more VAX Ada program units).

Table E–1: Description of Test Programs

Programs	Descriptions
BL6PERF0	250 lines (27 disk blocks). A set of eight library package specifications that were taken from the front end of a compiler written in Ada. The set includes units for lexical analysis and parsing.
BL6PERF1	850 lines (56 disk blocks). One of the package bodies that goes with BL6PERF0. The **with** clause names seven packages (five from BL6PERF0). The body includes one instantiation of INTEGER_IO.TEXT_IO.
ADABNF	3400 lines (284 disk blocks). A library package specification that provides the control tables for the LALR parser in BL6PERF0. This package specification consists of a small number of array objects initialized by very large aggregates. One of these is an array aggregate consisting of 2700 named associations in which each component value is itself a record aggregate with two components. Another is similar, with nearly 500 associations in which each component value is a record aggregate with three components.
ADAMACROS	125 lines (10 disk blocks). A complete program (library procedure) that performs a simple macro-like substitution for a set of files using a simple definition file. (Used to construct the VAX Ada-specific versions for the implementation-dependent tests of the Ada validation tests.)
STARLET0	29,000 lines (2113 disk blocks). Two library package specifications consisting of CONDITION_HANDLING and STARLET as found in the predefined library for VAX Ada.
STARLET1	6 lines (1 disk block). A very small procedure that uses a single named number from package STARLET.
NULLP	4 lines (1 disk block). A procedure consisting of only a null statement.

Each of these units was compiled using working sets of 500, 750, 1000, 1250, 1500, 1750, 2000, 2500, 3000, 4000, 6000, and 10000 pages.

For these experiments, the working set size was established using the following DCL command, where x is the desired working set size in pages:

```
$SET WORKING_SET/LIMIT=x/QUOTA=x/NOADJUST
```

This causes the working set to be fixed as specified and disables the VMS automatic adjustment strategies.

E.1.1.1 Effect of Working Set on Paging Rate

Page fault data is shown in Figure E–1. The vertical axis is the number of page faults times 1000. For the program STARLET0, the number of page faults is scaled by a multiplier of 10.

The significant feature of the graph in Figure E–1 is the working set associated with the knee of each curve. As shown, the knee occurs at around 2500 pages (1-1/4 megabytes) in each case.

The value of 2500 pages is significant because it indicates that as the working set is reduced below this value, the amount of paging rises rapidly. This increased paging translates into increased system load that affects the overall system performance and responsiveness for all users. On the other hand, as the working set is increased above this figure, the paging does not decrease very rapidly. Thus, a working set of 2500 pages should be considered as the minimum required for efficient VAX Ada compilations. (This figure is also reinforced by the analysis of compilation rates.)

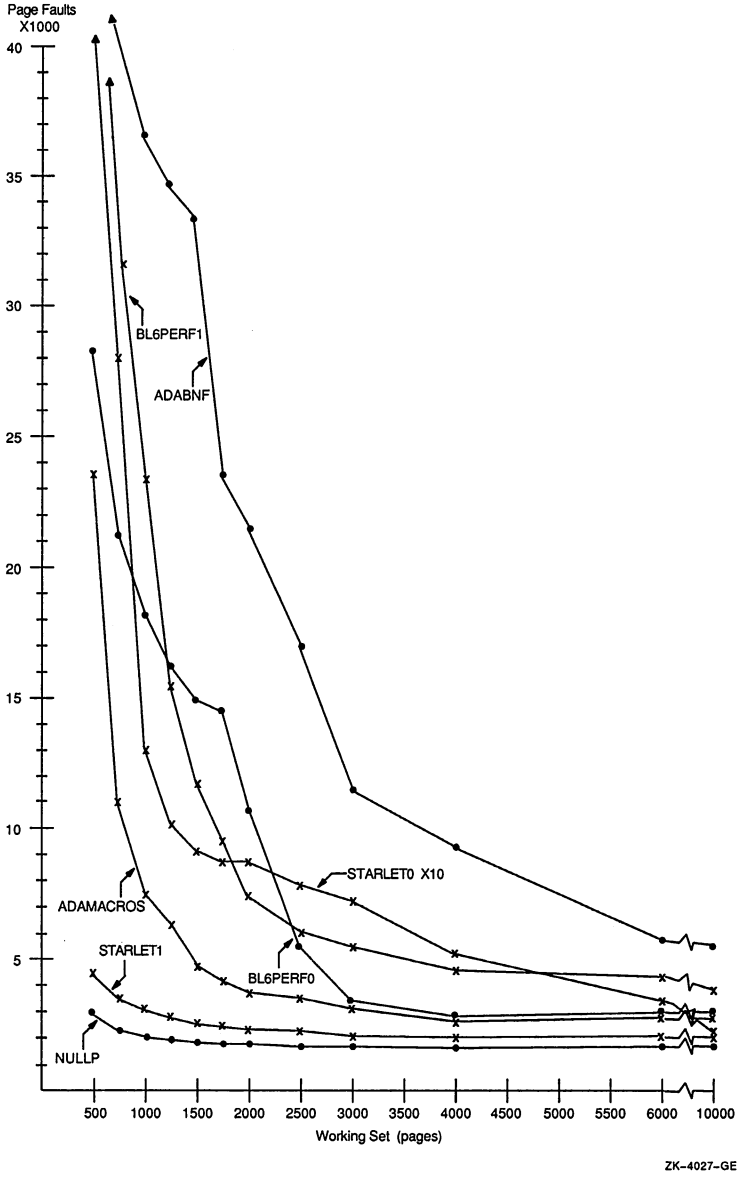
E.1.1.2 Effect of Working Set on Compilation Rate

Using data derived from the sample compilations used to analyze page faulting behavior, Figure E–2 shows the compilation rate (measured in source lines per elapsed minute) in relation to working set size. These compilation rates are based on the number of source lines compiled per elapsed minute and not on the number of source lines compiled per CPU minute, because the number of source lines per elapsed minute is a better measure of compilation rate throughput. The vertical axis in Figure E–2 is the number of lines compiled per elapsed minute, which is scaled by a multiplier shown on the graph for each program.

In Figure E–2, there is a clearly defined knee at 2500 pages, for which the following determinations can be made:

- At working sets above 2500 pages, there is very little increase in the compilation rate.
- At working sets below 2500 pages, there is a decrease in the compilation rate due to the paging overhead.

Figure E–1: Page Faults Versus Working Set Size



These measurements were made under standalone conditions on a system with a relatively large amount of memory, so that even at very small working set sizes, page faults could be quickly and cheaply satisfied from VMS cached pages in main memory. On a more heavily loaded system, or one with less main memory, the cost of paging is likely to be higher as more page faults require disk access. This increase in page faults would result in more sharply reduced compilation rates and throughput at the smaller working sets.

It may seem surprising that such a relatively large working set is desirable even for very small programs. Part of the explanation for this is the rather large size of the VAX Ada compiler itself—approximately 4600 disk blocks. The operation of just paging in the VAX Ada compiler during its execution phase causes more paging than other compilers. (Other VMS compilers supplied by Digital are generally in the 600- to 1200-block range.)

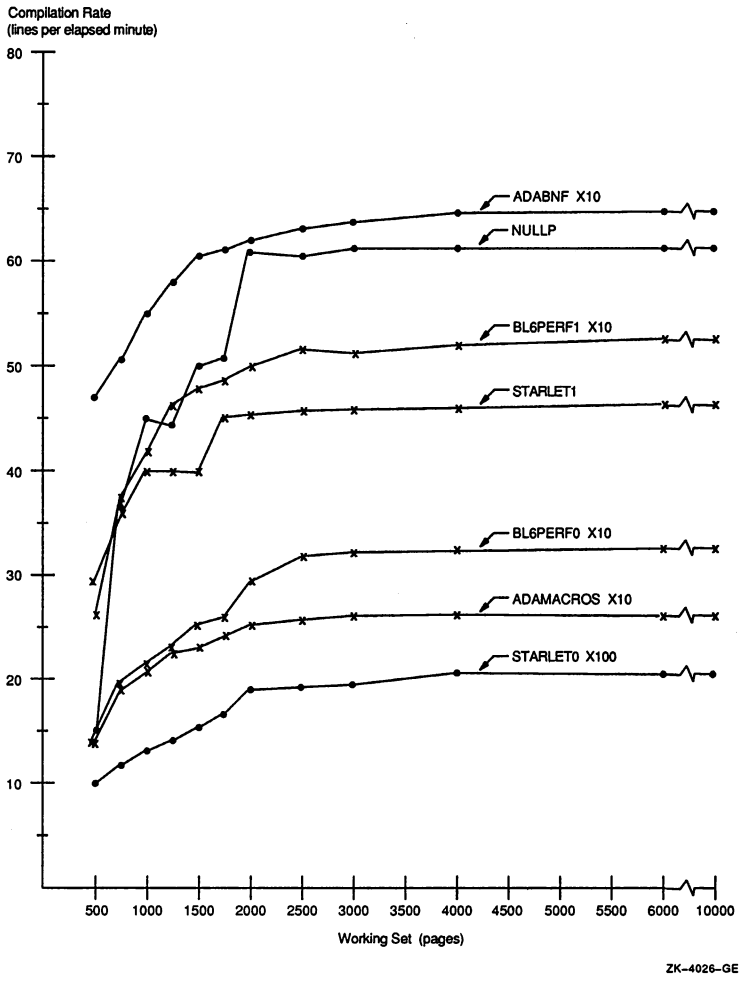
E.1.1.3 Suggestions for Controlling Working Set Sizes

Some VMS systems are used in environments where a large number of users coexist with relatively small working set quotas (an educational environment is a key example). In such settings, VAX Ada may appear both to be very sluggish for each individual user and to induce a large system overhead affecting all users. A suggested practice is to establish a special batch job queue intended primarily for VAX Ada compilations. (While especially important in such cases, this practice is a good one even on systems where there is generous memory for large user quotas.)

The characteristics of the queue should be set to allow a fairly large working set quota (at least 2500 pages) with a working-set extent of 4000 or more for each of the concurrent compilations operating at normal interactive priority (priority 4). In general, you can allow as many concurrent compilations as you have VAX units of performance (VUPs). (One VUP is equal to the performance of a VAX-11/780.)

These characteristics will allow VAX Ada compilations to complete efficiently and without inducing excessive system overhead. You may need to experiment a bit with the various parameters and options to find the configuration that works best on your system.

Figure E–2: Compilation Rate Versus Working Set Size



The ACS COMPILE and RECOMPILE commands are designed to support this style of batch processing. By default, the COMPILE and RECOMPILE commands submit compilations to the queue named by the logical name ADA$BATCH. If ADA$BATCH is not defined, then the system batch queue SYS$BATCH is used. When a separate queue for VAX Ada compilations is desired, ADA$BATCH should be defined as a system logical name whose translation is the name of the appropriate queue.

E.1.2 Virtual Address Space

It is useful to consider the peak virtual memory required for the compilation of each unit. This is the total virtual memory required for the execution of the process for all purposes, including the compiler itself, as well as dynamic memory used for representing the program, input-output buffers, and so on.

The virtual memory requirements of the programs compiled for the experiments on working set size are as follows:

Program	Peak Virtual Pages
BL6PERF0	9100
BL6PERF1	10300
ADABNF	12000
ADAMACROS	9100
STARLET0	18900
STARLET1	8900
NULLP	8900

These figures suggest that in a large-scale production environment the system and individual quotas should be configured to allow a virtual address space of at least 20,000 pages (10 megabytes) for VAX Ada compilations. In less demanding environments, 15,000 pages might generally be adequate. (The actual virtual memory used and remaining for a compilation is displayed as part of the summary in the compilation listing.)

See Section E.2 for more information on specific system parameters that affect the amount of available virtual address space.

E.2 Resource Requirements

The following sections provide information on the system resources used by the VAX Ada compiler and program library manager. Key System Generation Utility (SYSGEN) and User Authorization File (UAF) parameters are described, and suggested minimum values are noted. The parameters discussed are those for which VAX Ada requires larger values than the VMS defaults, or for which the values are larger than those required for other VAX languages.

The VMS operating system controls resource usage through two primary sets of parameters: system-wide parameters and per-process parameters. You use the SYSGEN utility to define and modify system-wide parameters (SYSGEN parameters). Per-process parameters are controlled on a per-user-name basis by quotas and defaults that are contained in the User Authorization File (UAF parameters); you use the Authorize Utility to modify UAF entries.

A complete description of VMS resources, SYSGEN, and UAF parameters is contained in the *VAX/VMS System Manager's Reference Manual*. You should be familiar with the procedures described there for modifying SYSGEN parameters. The recommended procedure is to add site-specific parameters to the file SYS$SYSTEM:MODPARAMS.DAT and then invoke the SYS$UPDATE:AUTOGEN command procedure.

E.2.1 ASTLM—AST Queue Limit Parameter

The UAF AST queue limit (ASTLM) parameter limits the sum of the following:

- The number of asynchronous system trap (AST) requests that a user's process can have outstanding at one time
- The number of scheduled wakeup requests that a user's process can have outstanding at one time

The VAX Ada delay statement is implemented as a call to the VMS SYS$SETIMR system service, which executes an AST routine. Also, VAX Ada provides facilities for calling VMS system routines that execute AST routines; for example, the package STARLET provides the system services SYS$QIO and SYS$QIOW, both of which allow you to specify an AST service routine. The routines in the package TASKING_SERVICES execute ASTs as well (although the execution and handling of the ASTs is hidden by the package).

The suggested VMS typical value for ASTLM is 24; this should be sufficient for most Ada programs. However, if your programs involve tasks that could execute many delay statements simultaneously, or you use a high number of calls to VMS system routines that execute AST routines, you may want to increase this value.

See the *VAX Ada Run-Time Reference Manual* for more information on ASTs in VAX Ada programs, especially programs that use tasks.

E.2.2 ENQLM—Enqueue Quota Parameter

The UAF enqueue quota (ENQLM) parameter limits the number of locks that a process and its subprocesses can own. VAX Record Management Services (RMS) uses locks to synchronize access to shared files and records.

VAX Ada and the VAX Ada program library manager use VAX RMS to access the VAX Ada program library index file, ADALIB.ALB. Up to eight locks may be used for each library or sublibrary that is open. When a sublibrary is opened, it is usually the case that its parent and its ancestors are also opened. Therefore, if you are using a sublibrary that has a parent and a grandparent, up to 24 (3 * 8) locks may be needed. Commands such as ACS COPY UNIT may have two libraries or sublibraries open at the same time, which doubles the number of locks.

The suggested VMS typical value for ENQLM is 30. This value is not sufficient for using the VAX Ada program library manager and sublibraries; the recommended value for VAX Ada users is 60.

E.2.3 FILLM—Open File Limit Parameter

The UAF open file limit (FILLM) parameter limits the number of files that a user's process can have open at one time. This limit includes the number of network logical links that can be active at the same time. (See Section E.2.7 for more information about logical links.)

The VAX Ada compiler and program library manager use the value of FILLM to limit the total number of files open at one time. If a compilation involves a large number of units, and FILLM is set too low to allow all the files involved to be opened at the same time, the compilation can take a long time. For example, if a unit depends, directly or indirectly, on 99 additional units, and FILLM is set to 20, the compilation will be slower than if FILLM had been set to 100.

FILLM is a pooled limit with a suggested typical value of 20. This value is not sufficient for most VAX Ada compilations; the recommended minimum value for VAX Ada users is 50. Note, however, that the value of FILLM must always be lower than the value of the SYSGEN channel count (CHANNELCNT) parameter. See Section E.2.8 for more information.

If you increase the value of the FILLM parameter, you may also want to increase the value of the UAF BYTLM parameter. The general rule for the relationship between these two parameters is that the value of BYTLM should be at least 100 times the value of FILLM.

E.2.4 PRCLM—Subprocess Creation Limit Parameter

The UAF subprocess creation limit (PRCLM) parameter limits the number of subprocesses that a user's process can create.

The VAX Ada program library manager creates a subprocess to run the linker or the VAX Ada compiler for the ACS LINK/WAIT, COMPILE/WAIT, LOAD/WAIT, and RECOMPILE/WAIT commands. Only one subprocess is created, and the creating process waits for the termination of the subprocess.

The suggested VMS typical value for PRCLM is 2; this should be sufficient for the VAX Ada program library manager as well.

E.2.5 TQELM—Timer Queue Entry Limit Parameter

The timer queue entry limit (TQELM) parameter limits the sum of the following:

- The number of entries that a user's process can have in the timer queue
- The number of temporary common flag clusters that a user's process can have

The VAX Ada delay statement is implemented as a call to the VMS SYS$SETIMR system service, which adds an entry to the timer queue.

The suggested VMS typical value for TQELM is 20; this value should be sufficient for most Ada programs. However, if your programs involve tasks that could execute many delay statements simultaneously, you may want to increase this value.

See the *VAX Ada Run-Time Reference Manual* for more information on the interaction of the TQELM parameter with Ada tasking programs.

E.2.6 Virtual Memory Usage

The Memory Usage section describes the VAX Ada compiler's requirements for virtual address space and working sets. Various SYSGEN and user parameters should be set so that a VAX Ada compilation can use up to 20,000 pages (10 megabytes) of virtual memory. Working set parameters (see the following sections) should be adjusted to provide good performance for such large virtual address spaces.

E.2.6.1 VIRTUALPAGECNT—Maximum Number of Virtual Pages Parameter

The SYSGEN VIRTUALPAGECNT parameter sets the maximum number of virtual pages that can be mapped for any one process. The VMS default (8192 pages, or 4 megabytes) is too small for the compilation of large VAX Ada programs; a value of 20,000 (10 megabytes) is recommended.

You should also make sure that the system paging file is large enough to accommodate processes with large page file quotas. A paging file size of 30,000 blocks is adequate for single users when the value of VIRTUALPAGECNT is 20,000.

E.2.6.2 PGFLQUOTA—Paging File Quota Parameter

The UAF paging file quota (PGFLQUOTA) parameter limits the number of pages that your process can use in the system paging file. In effect, it limits the amount of read/write working storage that the compiler can use. PGFLQUOTA should be set to a value consistent with VIRTUALPAGECNT.

The VMS typical value of 12800 for PGFLQUOTA is too small; a value of 17000 (VIRTUALPAGECNT—3000) is recommended.

You should also make sure that the system page file is large enough to accommodate processes with large page file quotas. The size of the paging file should be greater than the maximum paging file quota for an individual. If your value of PGFLQUOTA or the system's value of VIRTUALPAGECNT is too small, large Ada compilations will fail with this message:

```
%F, Insufficient virtual memory
```

E.2.6.3 System Paging File

The system paging file (SYS$SYSTEM:PAGEFILE.SYS) determines the amount of paging space available for system processes and VAX RMS global buffers. The paging file size should be greater than the maximum paging file quota (UAF PGFLQUOTA parameter) for an individual. It should also be larger than the virtual page count (SYSGEN VIRTUALPAGECNT parameter). For example, if the value of VIRTUALPAGECNT is 20,000 pages, then a paging file of at least 30,000 blocks is needed.

If the paging file size is not properly adjusted with respect to these two parameters, system processing could appear to be suspended, and the following messages will appear on the operator's console:

```
%SYSTEM-W-PAGEFRAG, Page file badly fragmented, system continuing
%SYSTEM-W-PAGECRIT, Page file space critical, system trying
    to continue
```

It is recommended that you use the Digital-supplied AUTOGEN command procedure operations to adjust the size of the system paging file after adjusting the values of VIRTUALPAGECNT and PGFLQUOTA. Note that AUTOGEN sometimes chooses a paging file size that is too low; you may need to explicitly specify a paging file size before invoking the AUTOGEN operation. See the *Guide to Setting Up a VMS System* for information on how to use AUTOGEN and how to specify parameter values before invoking AUTOGEN.

See Sections E.2.6.1 and E.2.6.2 for more Ada-related information on these parameters. See the *Guide to Setting Up a VMS System* for information on the VMS default value of PGFLQUOTA; see the *VMS System Generation Utility Manual* for more information on the VMS default value of VIRTUALPAGECNT.

E.2.6.4 WSQUOTA and WSEXTENT—Working Set Quota and Extent Parameters

The UAF working set quota (WSQUOTA) parameter specifies the maximum size to which a user's physical memory size can grow on a typically loaded system. In other words, the system guarantees the user that WSQUOTA physical pages will be available to the user's process.

The UAF working set extent (WSEXTENT) parameter specifies the maximum size to which a user's physical memory can grow, independent of system load. WSEXTENT should be greater than or equal to WSQUOTA. If WSEXTENT is greater than WSQUOTA, the VMS operating system will attempt to provide additional physical pages to a process that is page-faulting heavily. Thus, on a lightly loaded system, the user's working set can grow beyond WSQUOTA up to WSEXTENT.

The discussion in Section E.1 shows that a VAX Ada compilation should have a working set of 2500 pages or more. If a user is going to do VAX Ada compilations interactively or in a subprocess (COMPILE/WAIT), the user's UAF entry should specify WSQUOTA=2500.

Many sites may not permit such large working sets for interactive jobs. In this case, a batch queue should be established for VAX Ada compilations. The VAX Ada batch queue can define values for WSQUOTA and WSEXTENT that override the UAF values. Compilations done in the batch queue can have much larger working sets, resulting in better compile times and better use of system resources.

E.2.6.5 Batch Queue Parameters

A batch queue with large working set parameters should be provided for VAX Ada compilations. This batch queue minimizes the number of concurrent VAX Ada compilations so that each compilation completes efficiently, without inducing excessive system overhead.

You should define the logical name ADA$BATCH in the system logical name table. The VAX Ada program library manager submits ACS COMPILE and RECOMPILE commands to the queue named by ADA$BATCH; if ADA$BATCH is not defined, these jobs are submitted to SYS$BATCH.

A typical VAX Ada batch queue that is designed to handle two concurrent compilations should be initialized with the following values:

```
$INITIALIZE/QUEUE/BATCH-
/BASE_PRIORITY=4/JOB_LIMIT=2-
/WSQUOTA=2500/WSEXTENT=4000     ADA_BATCH

$ASSIGN/SYSTEM ADA_BATCH ADA$BATCH
```

E.2.6.6 WSMAX—Working Set Maximum Number of Pages Parameter

The SYSGEN working set maximum number of pages (WSMAX) parameter sets the maximum number of pages for any working set on a system-wide basis. The value of WSMAX should be as large as the maximum WSEXTENT value assigned to any user or batch queue. The VMS default WSMAX of 1024 pages is too small for VAX Ada; a value of at least 4000 is recommended.

E.2.7 Program Library Networking Effects

There are several SYSGEN and DECnet parameters (on both the local and remote nodes) that may affect the availability of compilation units or files accessed over DECnet. For example, every time a file is opened on a remote node, a temporary connection, called a logical link, is made from the local node to the remote node. The total number of logical links allowed at one time is controlled by DECnet and may be set by the Network Control Program (NCP) Utility with the following command:

```
NCP>  SET EXECUTOR MAX LINKS X
```

The X in this command is the maximum number of logical links. This number represents the system (not process) quota; each connection between two nodes deducts one from the quota total. When setting this value, note that both the Ada compiler and program library manager use the per-process FILLM (file and logical link limit) quota, not the system quota, to limit the

total number of open files at one time. Limiting the total number of files open at one time will also reduce the potential number of logical links created to the remote node: a logical link is only required for files that are not accessed locally.

Also note that the creation of a logical link may also involve the creation of a process on the remote node. Thus, you may need to increase (or at least monitor) the values of the SYSGEN parameters MAXPROCESSCNT and BALSETCNT to allow more processes to be created for network servers (FALs).

If you are expecting to access a number of files or compilation units over DECnet, you may want to increase the value of the UAF buffered input-output byte count limit (BYTLM) parameter on your system. The value of this parameter affects the efficiency of program library operations performed over DECnet. Other parameters may also have an effect; DECnet parameters are documented in the *Guide to DECnet-VAX Networking*.

After you have set or reset system parameters to accommodate the use of remote nodes, you may want to run the Digital-supplied AUTOGEN command procedure (SYS$UPDATE:AUTOGEN.COM) to recompute optimal values for related parameters.

E.2.8 Channel Count Parameters

The SYSGEN channel count (CHANNELCNT) parameter specifies the maximum number of channels for each process in a VMS system; the UAF file and logical link limit (FILLM) parameter specifies the maximum number of files that can be open at one time, including active network logical links.

The VAX Ada compiler and program library manager use the value of FILLM to limit the total number of open files at one time; they close files as necessary to avoid exceeding that value. The compiler and program library manager assume that the value of FILLM (which has a default value of 20) is less than the value of CHANNELCNT (which has a default value of 127). You may need to raise the value of FILLM to accommodate Ada compilations that involve a large number of compilation units; however, you should be careful not to raise the value of FILLM above the value of CHANNELCNT.

Note that the value of the FILLM parameter may interact with other system parameters; see Sections E.2.3 and E.2.7 for more information on the FILLM parameter.

Appendix F

Compile-Time Diagnostic Messages

VAX Ada diagnostic messages generated by the compiler are presented in this appendix. Some messages can occur for both the compiler and program library manager; these are listed both in this appendix and in Appendix G.

The messages are listed in alphabetical order by ident. The ident is followed by the message text and the severity level of the message.

Chapter 3 presents complete details on compiler message categories; that information is summarized in the following sections for convenience. The /[NO]WARNINGS qualifier for the DCL ADA and ACS LOAD, COMPILE, and RECOMPILE commands allows you to control the display or listing of warning and informational messages.

F.1 Diagnostic Message Format

The general format of a compile-time diagnostic message is as follows:

```
%Facility_code-Severity_code-Ident-Message_text
```

Facility_code
Is a four-letter code (ADAC) that identifies the VAX Ada compiler.

Severity_code
Is a letter (F, E, W, or I) that indicates the severity of the message. The meaning of these severity codes is discussed in Section F.2.

Ident
Is a name that uniquely identifies the message.

Message_text
Is a description of the event that has taken place. Italicized items in the message text in this appendix indicate items that are replaced with specific information when the message is actually generated. Pertinent references to the *VAX Ada Language Reference Manual* are included in the message text wherever possible. The references have the following form:

```
[LRM section_number(paragraph number)]
```

A plus sign (+) in a reference indicates that the reference is to Digital-supplied text (as opposed to Ada standard text).

F.2 Diagnostic Message Severity Codes

A VAX Ada compile-time diagnostic message contains one of four codes—F, E, W, I—which indicate the severity of the error:

- **F** indicates a fatal error. An F-level message indicates that the intended request cannot be executed.
- **E** indicates a user error. An E-level message is often supplemented with informational (I-level) messages that give additional information about the error.
- **W** indicates a warning message. A command may have performed some, but not all, of your request, and you may have to verify the command output.
- **I** indicates an informational message. An I-level message often contains supplementary information about a preceding or otherwise related E-level error. The VAX Ada compiler further classifies I-level messages into one of four categories. These categories are discussed in Section F.3.

F.3 VAX Ada Compiler Informational Messages

The VAX Ada compiler issues four kinds of informational (I-level) diagnostic messages:

- WEAK_WARNINGS indicate potential problems in a legal program—for example, a possible run-time error. Weak warnings are the only kind of informational diagnostics that are counted in the summary statistics given at the end of a compilation listing.

- SUPPLEMENTAL messages are associated with a W-level or E-level diagnostic. Such messages provide additional information about a diagnostic or indicate that some checks were not performed due to previous errors.
- COMPILATION_NOTES provide information about how the compiler translated a program. They do not warn you of a possible problem, nor are they related to a W-level or E-level diagnostic.
- STATUS diagnostics include some end-of-compilation statistics and other status messages in the compilation listing.

F.4 VAX Ada Compiler Diagnostic Messages

ACCDESIGOBJTYPE, *Access_type* designates objects of *type*

Informational - supplemental.

ACPTSAMEENTRY, Accept for *entity* is within another accept *source_location* for the same *kind_of_entity* [LRM 9.5(8)]

Error.

ACSSUBMITSPR, Internal program library manager error—please submit a Software Performance Report (SPR) for *ACS_version*

Fatal.

ADASUBMITSPR, Internal Ada compiler error—please submit a Software Performance Report (SPR) for *Ada_version*

Fatal.

ADDRESS_ZERO, Attribute evaluates to SYSTEM.ADDRESS_ZERO [LRM 13.7a.1, 13.7.2(3+)]

Warning.

ADRREPENTRYINOU, All parameters of a task entry with an address representation clause must be of mode 'in'; parameter *entity* for task entry *entity* is of mode 'in out' [LRM 13.1(1)]

Error.

ADRREPENTRYNOTI, An address representation clause for a task entry (*entity*) is not supported by this implementation [LRM 13.5(7+)]

Error.

ADRREPENTRYOUT, All parameters of a task entry with an address representation clause must be of mode 'in'; parameter *entity* for task entry *entity* is of mode 'out' [LRM 13.1(1)]

Error.

ADRREPNOSYS, No with clause for predefined package SYSTEM applies to this unit [LRM 10.1.1(4), 13.5(3)]

Error.

ADRREPOCCURS, An address representation clause must not follow an occurrence of a name for *entity* declared *source_location* [LRM 13.1(8)]

Error.

ADRREPUNITNOTIM, An address representation clause for a subprogram, package, or task unit (*entity*) is not supported by this implementation [LRM 13.5(7+)]

Error.

AGGRARRNAMPOS, Component associations of an array aggregate must be all named or all positional (except 'others') [LRM 4.3.2(3)]

Error.

AGGRCOMPUNKN, Component type of array type *entity* is unknown due to a prior error

Error.

AGGROTHSCONTEXT, Array aggregate is not in a context that allows an 'others' choice [LRM 4.3.2(4-8)]

Error.

AGGROTHSNAMED, Array aggregate is not in a context that allows an 'others' choice in combination with other named associations [LRM 4.3.2(6)]

Error.

AGGROTHSSTATIC, The applicable *entity* corresponding to 'others' is not static [LRM 4.3.2(3)]

Error.

AGGRSTRCHAR, *Character* has no visible declaration that matches the component type *entity* for the array type *entity* [LRM 4.2(5)]

Error.

AGGRSTRLAST, String subaggregate does not correspond to the last dimension of array type *entity* [LRM 4.3.2(2)]

Error.

AGGRSTRTYPE, Component *entity* of array type *entity* is not a character type [LRM 4.3.2(2)]

Error.

ALLOCINITLIM, An allocator must not include initialization for an object of limited *entity* [LRM 7.4.4(8)]

Error.

AMBIGEXP, Ambiguous expression; the required type is *entity*, but more than one possible meaning has this type [LRM 8.7]

Error.

AMBIGEXPDNAME, Prefix of expanded name (selected component) denotes multiple containing program units or accept statements [LRM 4.1.3(18)]

Error.

AMBIGRANGE, Ambiguous range; the required type is *entity*, but more than one possible meaning has this type [LRM 8.7]

Error.

AMBIGRESUNIV, Ambiguous expression in number declaration can be either of type {universal_integer} or {universal_real} [LRM 8.7]

Error.

AMBIGRSL, Ambiguity detected during overload resolution [LRM 8.7]

Error.

ANANOTRAN, Analysis data file *file_specification* must be written to a random access device

Fatal.

ANASUBMITSPR, Internal Ada compiler error in analysis data collection—please submit a Software Performance Report (SPR) for *Ada_version* and try compilation with the /NOANALYSIS_DATA qualifier

Fatal.

ARRAGGRTYPE, Error occurs in array aggregate for *entity*

Informational - supplemental.

ARRBOUCONNOT, Array bounds must be all constrained or all unconstrained [LRM 3.6(2)]

Error.

ARRDEFMULOBJ, An array type definition is allowed as a subtype indication only for a constant or a variable [LRM 3.2(9)]

Error.

ASSIGNLIM, Assignment is not available for limited types [LRM 5.2(1), 7.4.4(1)]

Error.

ASSIGNNERESTYP, Result *type* of *entity* is not the same as *type* of *entity* [LRM 5.2(1)]

Error.

ASTENTIGN, First parameter *entity* is not an 'in' parameter passed by VALUE; pragma AST_ENTRY ignored [LRM 9.12a]

Warning.

ATTRARGS, Arguments not legal with attribute *attribute_name*; arguments ignored

Error.

ATTRNOTSUPP, Attribute *attribute_name* is not supported by this implementation [LRM A]

Error.

ATTRONEARG, Only one argument can be specified for attribute *attribute_name*; additional arguments ignored

Error.

ATTRRANGEFIXEDT, Attribute RANGE is not allowed as the range of a fixed-point type definition [LRM 3.5.9(8-10), 3.6.2(7), 4.9(11,13), A]

Error.

ATTRRANGEFLOATT, Attribute RANGE is not allowed as the range of a floating-point type definition [LRM 3.5.7(10-12), 3.6.2(7), 4.9(11,13), A]

Error.

ATTRRANGEINTTYP, Attribute RANGE is not allowed as the range of an integer type definition [LRM 3.5.4(3-4), 3.6.2(7), 4.9(11,13), A]

Error.

ATTRUNKN, Attribute *attribute_name* is not known to this implementation

Error.

BADSLICE, A range for a slice operation must be the only operand [LRM 4.1.2(1-3)]

Error.

BAD_ALIGNMENT, Must be aligned on at least a *number* bit boundary [LRM 13.4(5+)]

Error.

BASETYPERANGEIN, Type of range is inconsistent with its context, which requires *entity* [LRM 8.7]

Error.

BASETYPERANGEUN, Type of range is unknown due to a prior error

Informational - supplemental.

BASICAFTLATER, A basic declaration is not allowed after later declarations, which begin *source_location* [LRM 3.9(2)]

Error.

BODYBPUNIT, A body declaration is not allowed for predefined *entity*

Error.

CANTRUN, Ada compiler cannot be invoked from a DCL RUN command, or incorrect command definition file (.CLD) used

Fatal.

CASEEXPDERGENTY, Expression result type *entity* is derived from generic formal type *entity* [LRM 5.4(3)]

Error.

CASEEXPGENTYPE, Expression result type *entity* is a generic formal type [LRM 5.4(3)]

Error.

CASESUBTNOTSTAT, *Subtype* is not static [LRM 5.4(4)]

Informational - supplemental.

CASETYPEAPPLIES, All values of the base type must be given unless the expression is the name of an object whose subtype is static, or a qualified expression or type conversion whose type mark denotes a static subtype [LRM 5.4(4,5)]

Informational - supplemental.

CASEUNIVINTOTHS, Expression result type *entity* implies that an 'others' alternative is required [LRM 5.4(4)]

Error.

CHARSTRTYPE, *Entity* has no visible declaration that matches the component type *entity* for the string type *entity* [LRM 4.2(5)]

Error.

CHOICE_OVERLAP*, Choice *value_or_range* overlaps another choice for *value_or_range source_location*

Error. The given value or range overlaps with the value or range of another choice of the same case statement, variant part, or aggregate. There are multiple variations of this message depending on whether each of the overlapping choices is a single value or range of values, and whether the overlap is a single value or a range of values. If the overlap occurs within the choices of a single alternative (separated by vertical bars (|)), then the source location is given as "in this same sequence." Otherwise, appropriate variants of the message occur twice: as an informational message on the first of the overlapping pair of choices, and as an error message on the second of the pair of choices.

CIRCTYPE, Circular type declaration [LRM 3.3(8)]

Error.

CL_ACCLIBDEN, Access to program library *directory_specification* is denied due to file protections associated with the VMS directory or the library index file

Error.

CL_ADDED, *Entity* added to library *file_specification*

Informational - status.

CL_ADDED_1, Replaces older version compiled *date_time*

Informational - status.

CL_ADDED_2, Supersedes *entity* compiled *date_time*

Informational - status.

CL_ADDED_3, Corresponds to *entity* compiled *date_time*

Informational - status.

CL_BPUNOTLIB, *Entity* depends on *entity*, which is a predefined unit in this library

Error.

CL_COMPLETED, *Entity* completed in library *file_specification*; from instantiation *entity*

Informational - status.

CL_ERROPELIB, Error opening library for compilation

Fatal.

CL_INVACUFMT, The compilation unit file (.ACU) file has an invalid format

Error.

CL_INVLIBFMT, The library index file (ADALIB.ALB) has an invalid format

Error.

CL_NEWLIB, Program library *directory_specification* was created or last converted by a newer *compiler* version (*version-number*) and cannot be used by the current *compiler* version (*version-number*)

Error.

CL_NEWUNIT, *Entity* was compiled by a newer compiler (*version-number*) and cannot be used by the current compiler (*version-number*)

Error.

CL_NOTADDED, *Entity* in file *file_specification* was not added to library

Informational - status.

CL_OBSLIB, Program library *directory_specification* was created or last converted by an obsolete *compiler* version (*obs_version*) and cannot be used by the current *compiler* version (*version-number*); use the ACS CONVERT LIBRARY command to convert the library to the current version

Error.

CL_OBSUNIT, *Entity* was compiled by an obsolete compiler (*version-number*) and cannot be used by the current compiler (*version-number*); this unit should be compiled using the current compiler

Error.

CL_OLDREFBPU, *Entity* depends on predefined *name*, which has been redefined

Error.

CL_OLDREFUNI, *Entity* depends on *entity*, which was recompiled *date_time*

Error.

CL_PREV20, Program library *directory_specification* was created by a VAX Ada version prior to Version 2.0 and cannot be used by the current *compiler* version (*version-number*); use the ACS CONVERT LIBRARY command to convert the library to the current version

Error.

CL_SOURCEFILE, *Entity* originated in source file *file-spec*

Informational - supplemental.

CL_SPECOMLAT, *Entity* now depends upon its specification (which was compiled later into the library) and must be (re)compiled

Error.

CL_SPENOTFOU, Specification for *name* not found in library

Error.

CL_STUNOTFOU, *Name* contains no body stub for *name*

Error.

CL_TGTMISMATCH1, *Name* is compiled for an unknown target [LRM 13.7(11)]

Error.

CL_UNIALREXI_1, Unit *name* already exists in the library and was not replaced

Error.

CL_UNIALREXI_2, *Entity* was previously compiled into the library from the same source file *file_specification* and was not replaced

Informational - status.

CL_UNIDEPDFL, *Name* depends on pragma LONG_FLOAT(D_FLOAT), but that is not the library definition [LRM 3.5.7a]

Error.

CL_UNIDEPGFL, *Name* depends on pragma LONG_FLOAT(G_FLOAT), but that is not the library definition [LRM 3.5.7a]

Error.

CL_UNIDEPMEM, *Name* depends on SYSTEM.MEMORY_SIZE, which has been redefined [LRM 13.7(11)]

Error.

CL_UNIDEPNAM, *Name* depends on SYSTEM.SYSTEM_NAME, which has been redefined [LRM 13.7(11)]

Error.

CODESTMTAGGR, An expression in a code statement must be a compile-time-constant record aggregate [LRM 13.8(2,4)]

Error.

CODESTMTDECL, Declarations in *entity* must be either use clauses or pragmas [LRM 13.8(3)]

Error.

CODESTMTHNDLR, Exception handlers may not be specified in *entity* [LRM 13.8(3)]

Error.

CODESTMTINHNDLR, A code statement may not appear in an exception handler [LRM 13.8(3)]

Error.

CODESTMTPOSI, A code statement must appear in the sequence of statements of a procedure body [LRM 13.8(3)]

Error.

CODESTMTPROC, *Entity* may not include code insertions as it is not a procedure body [LRM 13.8(3)]

Error.

CODESTMTSTMT, Statements in *entity* must be either code insertions or pragmas [LRM 13.8(3)]

Error.

CODESTMTTYPE, *Entity* is not declared in the predefined library package MACHINE_CODE [LRM 13.8(4)]

Error.

COMPASSDUP, Duplicate value given for *entity* [LRM 3.7.2(4), 4.3(6)]

Error.

COMPASSEXPPRIOR, Error occurs in expression that corresponds to *entity*

Informational - supplemental.

COMPASSNOASSOC, No value given for *entity* [LRM 3.7.2(4), 4.3(6)]

Error.

COMPASSNOTCOMP, *Entity* is not a component of *entity* [LRM 4.3.1(1)]

Error.

COMPASSNOTDISCR, *Entity* is not a discriminant of *entity* [LRM 3.7.2(4)]

Error.

COMPASSNOTUSED, Excess association has no corresponding component [LRM 4.3(6)]

Error.

COMPASSTYPENETY, Type *entity* of *entity* is not the same as type *entity* of *entity* [LRM 3.7.2(4), 4.3.1(1), 12.3.2(3)]

Error.

COMPASSWRONGVAR, *Entity* is in the wrong variant [LRM 4.3.1(2)]

Error.

COMPDECLACC, Illegal component declaration; found access type definition when expecting a subtype indication [LRM 3.7(2)]

Error.

COMPDECLARRAY, Illegal component declaration; found array type definition when expecting a subtype indication [LRM 3.7(2)]

Error.

COMPDECLREC, Illegal component declaration; found record type definition when expecting a subtype indication [LRM 3.7(2)]

Error.

COMPDEPSGENFORM, A component representation clause is not allowed for a component whose type *entity* is, or depends on, a generic formal type [LRM 13.1(10+,14), F.4]

Error.

COMPLINCOMPL, Completion failed because of one or more incomplete generic bodies

Error.

COMP_OVERLAP, Storage place for this record component overlaps that allocated to *component* [LRM 13.4(7)]

Error.

COMP_SIZE_NOT_S, Size of component is not constant [LRM 13.4(7)]

Error.

CONFBASETYPE, Type *entity* of *entity* is not the same as type *entity* of *entity* [LRM 6.3.1]

Error.

CONFDENOT, *Entity* is not the same as *entity*, which is denoted *source_location* [LRM 6.3.1]

Error.

CONFDFLTEXP, A default expression is given for this *name* or *entity*, but not both [LRM 6.3.1]

Error.

CONFDFLTIN, Default mode 'in' applies for this *entity*, but 'in' is explicit for *entity* [LRM 6.3.1]

Error.

CONFDIFFNAME, Name *name* is not the same as *entity* [LRM 6.3.1]

Error.

CONFEXPLIN, Explicit mode 'in' is given for this *name*, but 'in' is implicit for *entity* [LRM 6.3.1]

Error.

CONFLEX, *Identifier_or_literal* is not the same as *syntactic_form* that occurs *source_location* [LRM 6.3.1]

Error.

CONFLITVAL, Literal *literal* does not have the same value as *entity* [LRM 6.3.1(2)]

Error.

CONFMODE, The mode of *name* is not the same as the mode of *entity* [LRM 6.3.1]

Error.

CONFMULTI, The multiple declaration form of *name* does not match *entity* [LRM 6.3.1]

Error.

CONFNUMASSOC, The number of associations is not the same as *source_location* [LRM 6.3.1]

Error.

CONFNUMCHOICE, The number of choices is not the same as *source_location* [LRM 6.3.1]

Error.

CONFNUMOBJ, The number of discriminants or formals is not the same as *source_location* [LRM 6.3.1]

Error.

CONFSINGLE, The single declaration form of *name* does not match *entity* [LRM 6.3.1]

Error.

CONFTYPECONVDER, A type conversion corresponding to an 'in out' or 'out' parameter of *entity* is not allowed [LRM 6.3.1]

Error.

CONFWITH, Error detected during conformance check with *entity*

Informational - supplemental.

CONFWITHTYPEMAR, Error detected during conformance check with type mark of *entity*

Informational - supplemental.

CONSTNOINITGFPT, Constant *name* of *type_or_subtype* is not initialized; *type_name* is a generic formal private type [LRM 3.2.1(2)]

Error.

CONSTNOINITNPT, Constant *name* of *type_or_subtype* is not initialized; *type_name* is not a private type [LRM 3.2.1(2), 7.4(1,3)]

Error.

CONSTRAINTS, CONSTRAINT_ERROR will be raised here [LRM 4.3.1(3), 4.3.2(11), 5.2(4), 6.4.1(10), 12.3.4(5)]

Informational - weak warning.

CONSTRINCOSEP, A constraint is not allowed for *entity* whose corresponding full type does not occur in the same private part as that incomplete type [AI-00007]

Error.

CONSTRNOTSTATIC, Constraint *source_location* for *entity* is not static [LRM 13.2(6), 13.4(7), 13.9a.2]

Error.

CONSTRPREV, Error occurs in apparent index or discriminant constraint for *entity*

Informational - supplemental.

CONSTRPREVDESIG, Error occurs in apparent index or discriminant constraint for *entity*, which designates objects of *type*

Informational - supplemental.

DECLINPRAG, Illegal declaration appears as a pragma argument; pragma ignored

Error.

DECLINSTMT, Ignored declaration appearing within a list of statements

Error.

DECLPRIPART, Illegal declaration in the private part of a package specification [LRM 7.1(2)]

Error.

DECLTASKSPEC, Illegal declaration in a task specification [LRM 9.1(3)]

Error.

DECLVISPART, Illegal declaration in the visible part of a package specification [LRM 7.1(2)]

Error.

DEFCONSTINPACK, A deferred constant declaration is allowed only in the visible part of a package specification [LRM 7.4(1)]

Error.

DEFEXPIGN, A default expression is only allowed for 'in' formal parameters; expression ignored [LRM 6.1(4), 12.1.1(2)]

Error.

DEFTYPEDIFFPACK, *Type* of deferred constant *name* is not declared immediately within this same package [LRM 7.4(4)]

Error.

DEPSYSNAMESTAB, Dependence established on predefined SYSTEM_NAME value of *literal*

Informational - compilation note.

DEPSYSNAMINCO, *Construct* is not allowed in combination with predefined SYSTEM_NAME value of *value* [LRM 13.7(6+)]

Error.

DERIVDERIV, Derivation from derived type *entity* before the end of the package visible part is not allowed [LRM 3.4(15), 7.4.1(4)]

Error.

DESCR_LENGTH_TO, CONSTRAINT_ERROR will be raised during descriptor evaluation [LRM 11.1(5), 13.9a.1.2]

Warning.

DIGTOOBIG, Digits expression value *number* exceeds the implemented range of *number* [LRM 3.5.7(8+), 13.7.1(4+)]

Error.

DIGZERNEG, Digits expression value *number* is less than 1 [LRM 3.5.7(3)]

Error.

DISCONSTRPREV, Error occurs in discriminant constraint for *entity*

Informational - supplemental.

DISCONSTRPREVDE, Error occurs in discriminant constraint for *entity*, which designates objects of *type*

Informational - supplemental.

DISCRIMIGN, A discriminant part can only be specified for a private, record, or incomplete type declaration; discriminants ignored [LRM 3.3.1]

Error.

DISCRIMIGNGEN, A discriminant part can only be specified for a private type declaration in a generic part; discriminants ignored [LRM 12.1.2]

Error.

DISCRIMUSE, *Discriminant_entity* is allowed only as an index or discriminant constraint value, or in a record component default expression [LRM 3.7.1(6)]

Error.

DISCRIMVARGEN, Type *entity* of discriminant *entity* is a generic formal type [LRM 3.7.3(3)]

Error.

DISCSPECACC, Illegal discriminant specification; found access type definition when expecting a type or subtype name [LRM 3.7.1(2)]

Error.

DISCSPECARRAY, Illegal discriminant specification; found array type definition when expecting a type or subtype name [LRM 3.7.1(2)]

Error.

DISCSPECREC, Illegal discriminant specification; found record type definition when expecting a type or subtype name [LRM 3.7.1(2)]

Error.

DUPBODY, *Entity* corresponds to *entity*, which already has a body given *source_location* [LRM 6,3(3)]

Error.

DUPBODYINST, A *kind* is not allowed corresponding to *entity* [LRM 12(2), 12.3(2,5)]

Error.

DUPEXCPUSE, Exception *entity* is already named *source_location* in this sequence of handlers [LRM 11.2(5)]

Error.

DUPSUBBODY, *Entity* has the same parameter and result type profile as *entity*, which already has a body given *source_location* [LRM 6.3(3)]

Error.

ENTITYIS, *Entity* is of *type_or_subtype*

Informational - supplemental.

ENTRYAFTREP, *Entity* is not allowed following a representation clause *location*

Error.

ENTRYCONTTASK, *Entity* is not an entry of the containing task unit *entity* [LRM 9.5(4)]

Error.

ENTRYCONTTASK1, *Entity* is not the containing task unit *entity* [LRM 9.5(4)]

Error.

ENTRYINTASKSPEC, An entry declaration is allowed only in a task specification [LRM 9.1(3), 9.5(1)]

Error.

ENTRYNOACCEPT, *Entity* has no corresponding accept statement in this task body

Informational - weak warning.

ENTRYOBJ, *Entity* must be an entry of a task object [LRM 8.5(9), 9.5(4)]

Error.

ENUMREPBADORD, The representation *entity* (*number*) is not greater than its predecessor (*number*) [LRM 13.3(4)]

Error.

ENUMREPDERIVED, An enumeration representation clause is not allowed for derived *entity* whose parent *entity* has derivable subprograms (such as *entity*) [LRM 13.1(3), 13.3]

Error.

ENUMREPDUP, An enumeration representation clause for *entity* is already given *source_location* [LRM 13.1(3)]

Error.

ENUMREPNOREP, No representation value is specified for *entity* [LRM 13.4(4)]

Error.

ENUMREPNOTLIT, *Entity* is not an enumeration literal [LRM 13.3(4)]

Error.

ENUMREPONECHO, Only one choice is allowed in an enumeration representation clause [LRM 13.3(4)]

Error.

ENUMREPREPDUP, The representation for *entity* is already given in this list [LRM 13.3(4)]

Error.

ENUMREPTOOMANY, Too many representation values are given for *entity* [LRM 13.3(4)]

Error.

ERRCOMPILE, Errors compiling *unit_name* in file *file_specification*

Error.

ERRORLIMIT, Terminating compilation because ERROR_LIMIT=*number* reached

Error.

ERRRECOMPILE, Errors recompiling *unit_name* in file *file_specification*

Error.

ERRSUBMITSPR, Internal Ada compiler error—please submit a Software Performance Report (SPR) for *Ada_version* and try compilation with a different value of /ERROR_LIMIT or use /NOERROR_LIMIT

Fatal.

EXITTARNG, An exit from loop *entity* is not allowed because of the intervening *entity* [LRM 5.7(3)]

Error.

EXPNOTSAFE, Value *number* is not in the range of safe numbers of any floating-point type [LRM 3.5.7(12), F.9.3]

Error.

EXPNOTSTATIC, Expression is not static [LRM 4.9]

Error.

EXPNOTSTATICEXC, Apparent static expression raises an exception during evaluation; assume nonstatic [LRM 4.9(2)]

Error.

EXPOPMIXED, An operand of exponentiation must not be another exponentiation (without separating parentheses) [LRM 4.4(2)]

Error.

FIRSTBIT_INVALI, Not a valid first bit position (too large or negative) [LRM 13.4(5), F.9.5]

Error.

FIRSTNAMEDSUBT, *Entity* is not a first named subtype [LRM 13.1(3)]

Error.

FIRST_NOT_1, *Entity* cannot be passed by S or UBS descriptor; the lower bound is not static with position equal to 1 [LRM 13.9a.1.2]

Warning.

FIXTYPERANGE, A fixed-point type declaration requires a range constraint [LRM 3.5.9(3)]

Error.

FORMACPTNAME, Illegal form for a name in an accept statement [LRM 9.5(2)]

Error.

FORMCHOICOMPASS, Illegal form for a choice in a component association; only an identifier is allowed [LRM 4.3(5)]

Error.

FORMDESIG, Illegal form for a designator [LRM 6.1(2)]

Error.

FORMEXP, Illegal form for an expression [LRM 4.4(2)]

Error.

FORMGENTYPE, Illegal form for a generic formal type [LRM 12.1(2), 12.1.2]

Error.

FORMNAME, Illegal form for a name [LRM 4.1(2)]

Error.

FORMNAMEOBJ, Illegal form for the name of an object [LRM 2.3, 3.2, 4.1(2)]

Error.

FORMNAMEPROC, Illegal form for the name of a procedure [LRM 6.1(2)]

Error.

FORMNAMERECREPA, Illegal form for the prefix of record representation attribute *attribute_name*; only a selected component is allowed [LRM 13.7.2(7-10)]

Error.

FORMNAMESTUB, Illegal form for the name of a body stub declaration; only an identifier is allowed [LRM 10.1(3)]

Error.

FORMNAMEUSECONT, Illegal form for a name in a use clause of a unit context clause; only an identifier is allowed [LRM 8.4(2), 10.1.1(2)]

Error.

FORMNAMEWITH, Illegal form for a name in a with clause; only an identifier is allowed [LRM 10.1.1(2)]

Error.

FORMNOTEXP, *Syntactic_form* is not a form of expression [LRM 4.4(2)]

Error.

FORMNOTEXPATTRF, An attribute function name is not a form of expression; an argument is required [LRM 4.1.4, 4.4(2), A]

Error.

FORMNOTNAME, *Syntactic_form* is not a form of name [LRM 4.1(2)]

Error.

FORMNOTNAMEOBJ, *Syntactic_form* is not a form of object name [LRM 4.1(2)]

Error.

FORMNOTRANGE, *Syntactic_form* is not a form of range [LRM 3.5(2)]

Error.

FORMNOTSUBTYPEI, *Syntactic_form* is not a form of subtype indication [LRM 3.3.2(2)]

Error.

FORMNOTTYPEMARK, *Syntactic_form* is not a form of type mark [LRM 3.3.1(2), 3.3.2(2)]

Error.

FORMRANGE, Illegal form for a range [LRM 3.5(2)]

Error.

FORMRANGGENARR, Illegal form for a range constraint in a generic formal array type [LRM 12.1.2(2)]

Error.

FORMRANGNUMERTY, Illegal form for a range in a numeric type definition; only a pair of expressions separated by ".." is allowed [LRM 3.5(2), 3.5.4(3-4), 3.5.7(10-12), 3.5.9(8-10)]

Error.

FORMRANGSUB, Illegal form for a type mark or range in a discrete range or scalar subtype indication [LRM 3.3.2(2), 3.5(2)]

Error.

FORMRANGTYPE, Illegal form for a type mark or range in a membership test [LRM 4.4(2), 4.5.2(10)]

Error.

FORMSUBAGGR, Illegal form for a subaggregate [LRM 4.3.2(2)]

Error.

FORMSUBTYPEIND, Illegal form for a subtype indication [LRM 3.3.2(2)]

Error.

FORMSUBTYPEINDG, Illegal form for a subtype indication in a generic formal part; only a type mark is allowed [LRM 12.1(2,4)]

Error.

FORMTYPEDEF, Illegal form for a type definition [LRM 3.3.1(2), 12.1(2)]

Error.

FORMTYPEMARK, Illegal form for a type mark [LRM 3.3.2(2)]

Error.

FOUNDEXP, Found *lexical_element* when expecting *lexical_element*

Error.

FOUNDEXPID, Found identifier when expecting parameter specification

Error.

FOUNDEXPLABEL, Found statement label when expecting one of { "accept" "delay" "pragma" "terminate" "when" }

Error.

FOUNDEXPNULL, Found reserved-word "null" when expecting one of { "accept" "delay" "pragma" "terminate" }

Error.

FRAME_ALIGN, The maximum alignment for stack objects is 4 (longword) [LRM 13.4(4+)]

Warning.

FRAME_TOO_BIG, The storage allocated in this frame exceeds the implementation limit of *number* bytes [LRM F.9.5]

Warning.

FULLTYPEDISCRIM, The full type must be a record type because the corresponding *entity* has discriminants [LRM 7.4.1(3)]

Error.

FULLTYPEINCOPRI, Full type for *entity* must not itself be an incomplete or private type [LRM 3.8.1, 7.4.1]

Error.

FULLTYPELIM, Full type *name* is limited, but corresponding *entity* is not limited [LRM 7.4.1(3)]

Error.

FULLTYPESUBPOUT, The full type for limited private type *entity* that occurs in an 'out' parameter must not be limited [LRM 7.4.4(4)]

Error.

FULLTYPEUNCARR, The full type for *entity* is an unconstrained array type [LRM 7.4.1(3)]

Error.

FULLTYPEUNCREC, The full type for *entity* is an unconstrained type with discriminants [LRM 7.4.1(3)]

Error.

FUNCCALLWOPARAM, *Entity* is not callable without any parameters

Informational - supplemental.

FUNCMODEIN, A formal parameter of a function declaration must have mode 'in' (explicit or default) [LRM 6.5(1)]

Error.

FUNCNORET, *Entity* does not include any return statements [LRM 6.5(1)]

Error.

GCS_NOSHARE_BEC, This instantiation will not use a shareable generic specification or body because *entity* is not yet supported

Warning.

GENACNOCONSTDF, *Entity* is neither constrained nor a type with discriminants with defaults [LRM 12.3.2(4), AI-00037]

Error.

GENACNOCONSTDF1, The use of corresponding formal *entity* within *entity* requires a constrained actual subtype or a type with discriminants with defaults

Informational - supplemental.

GENACTACCDESCON, Designated *entity* for access *entity* is constrained *source_location*, but designated *entity* for *entity* is not constrained [LRM 12.3.5(1)]

Error.

GENACTACCDESTYP, Designated *entity* for actual *entity* is not the same as designated *entity* for generic formal *entity* [LRM 12.3.5(1)]

Error.

GENACTACCESS, Actual *entity* corresponding to generic formal *entity* is not an access type or subtype [LRM 12.3.5(1)]

Error.

GENACTARRAY, Actual *entity* corresponding to generic formal *entity* is not an array type or subtype [LRM 12.3.4(1)]

Error.

GENACTARRCOMCON, Component *entity* for array *entity* is constrained *source_location*, but component *entity* for *entity* is not constrained [LRM 12.3.4(4)]

Error.

GENACTARRCOMTYP, Component *entity* for actual *entity* is not the same as component *entity* for generic formal *entity* [LRM 12.3.4(4)]

Error.

GENACTARRCONSTR, Generic formal *entity* is constrained, but actual *entity* is not constrained [LRM 12.3.4(2)]

Error.

GENACTARRINDNUM, Generic formal *entity* and actual *entity* do not have the same number of dimensions [LRM 12.3.4(2)]

Error.

GENACTARRINDTYP, At index position *number*, generic formal *entity* has index *entity* while actual *entity* has index *entity* [LRM 12.3.4(3)]

Error.

GENACTARRUNCON, Generic formal *entity* is not constrained, but actual *entity* is constrained *source_location* [LRM 12.3.4(2)]

Error.

GENACTDFLTUNKN, Default actual is unknown because of a prior error related to *entity* of *entity*

Informational - supplemental.

GENACTDISCR, Actual *entity* corresponding to generic formal *entity* is not a discrete type or subtype [LRM 12.3.3(1)]

Error.

GENACTEXCNUM, The number of actual parameters exceeds the number of formals in instantiation of *entity* [LRM 12.3(3-4)]

Error.

GENACTFIXED, Actual *entity* corresponding to generic formal *entity* is not a fixed-point type or subtype [LRM 12.3.3(1)]

Error.

GENACTFLOAT, Actual *entity* corresponding to generic formal *entity* is not a floating-point type or subtype [LRM 12.3.3(1)]

Error.

GENACTINT, Actual *entity* corresponding to generic formal *entity* is not an integer type or subtype [LRM 12.3.3(1)]

Error.

GENACTLIMITED, Actual *entity* is limited, but the corresponding generic formal *entity* is not limited [LRM 12.3.2(2)]

Error.

GENACTNOCONST1D, Generic formal *entity* of *entity* does not allow an unconstrained actual array type or an unconstrained actual type with discriminants (with or without defaults)

Informational - supplemental.

GENACTNOCONST1U, Generic formal *entity* does not allow an unconstrained actual array type or an unconstrained actual type with discriminants (with or without defaults)

Informational - supplemental.

GENACTNOCONST2, Corresponding formal *entity* is used within *entity* as an actual corresponding to formal *entity*

Informational - supplemental.

GENACTNOTCONSTR, *Entity* is not constrained [LRM 12.3.2(4), 13.7a.1, 13.10.2(2+)]

Error.

GENACTNUMDISCRI, The number of discriminants in actual *entity* does not equal the number of discriminants in generic formal *entity* [LRM 12.3.2(3)]

Error.

GENACTOBJINLIM, The actual corresponding to *entity* must not be of *limited_type* [LRM 7.4.4(9), 12.3.1(3)]

Error.

GENACTPREV, Error occurs in actual corresponding to generic formal *entity*

Informational - supplemental.

GENACTPREVOBJ, Error occurs in actual corresponding to *entity*

Informational - supplemental.

GENACTPRIVCONST, *Entity* is constrained *source_location* [LRM 12.3.2(3)]

Error.

GENASSNORECSTDF, *Entity* is neither constrained nor a type with discriminants with defaults; compiled assuming generic body for *entity* does not require such a type for *entity*

Informational - compilation note.

GENASSNOREQCST, *Entity* is not constrained; compiled assuming generic body for *entity* does not require such a type for *entity*

Informational - compilation note.

GENASSOCMATCH, Association matches more than one generic formal of *entity*, including *entity* and *entity* [LRM 12.3(3)]

Error.

GENASSUMENOPACK, Compiled assuming no generic body for *entity*

Informational - compilation note.

GENCIRCINST, Recursive generic instantiation [LRM 12.3(18)]

Error.

GENDISCRIMDFLT, A default expression is not allowed for a discriminant in a generic formal private type [LRM 12.1.2(3)]

Error.

GENINFORMALLIM, *Entity* of generic formal 'in' object is limited [LRM 12.1.1(3)]

Error.

GENNOACTUAL, No actual (explicit or default) is given for generic formal *entity* in instantiation of *entity* [LRM 12.3(3)]

Error.

GENNOMATCH, *Name* is not the name of any formal parameter of *entity* [LRM 12.3(3)]

Error.

GENPAREXP, Illegal generic type definition; found expression when expecting <> [LRM 12.1(2)]

Error.

GENPARREC, Record type definitions are not allowed in generic parameter declarations; perhaps a private type was intended [LRM 12.1(2)]

Error.

GENPARSUB, Illegal generic parameter declaration; a subtype declaration is not allowed [LRM 12.1(2)]

Error.

GENPRIORACT, An actual parameter for *name* is already given in this instantiation of *entity* [LRM 12.3(3)]

Error.

GENTYPENONSTATI, *Entity* is, or is derived from, a generic formal type

Informational - supplemental.

GOTOTARNG, Illegal transfer to *entity* (for example, into compound statement or outside body) [LRM 5.9(3)]

Error.

IGNORE, Ignored *lexical_element*

Error.

IGNOREDECL, Declaration ignored due to syntactic errors within it

Informational - supplemental.

IGNOREDECLL, Declaration ignored due to syntactic errors

Informational - supplemental.

IGNOREPARENS, Empty parentheses ignored

Error.

IGNOREUNEXP, Unexpected *lexical_element* ignored

Error.

IMPT_VAL_0, *Entity* is not a string literal of 31 characters or less; SYSTEM.IMPORT_VALUE result is undefined [LRM 13.7a.8]

Warning.

INCONEXP, Inconsistent expression; the required type is *entity*, but no possible meaning has this type [LRM 8.7]

Error.

INCONOFULLTYPE, *Name* has no corresponding full type declaration [LRM 3.8.1(3)]

Error.

INCONRESUNIV, Inconsistent overload resolution; the result type is neither {universal_integer} nor {universal_real} [LRM 3.2.2(1), 8.7]

Error.

INCONRSL, Inconsistency detected during overload resolution [LRM 8.7]

Error.

INDCONSTRPREV, Error occurs in index constraint for *entity*

Informational - supplemental.

INDCONSTRPREVDE, Error occurs in index constraint for *entity*, which designates objects of *type*

Informational - supplemental.

INDCONSTRTOOFEW, Index constraint has too few constraints [LRM 3.6.1(1)]

Error.

INDCONSTRTOOMAN, Index constraint has too many constraints [LRM 3.6.1(1)]

Error.

INDEXESNOTSTATI, Index subtypes for *entity* are not static [LRM 4.9(11)]

Error.

INDXENTRONEARG, An indexing of an entry family must have exactly one argument [LRM 9.5(2)]

Error.

INDXPREV, Error occurs in indexing of *entity* of *type*

Informational - supplemental.

INDXTOOFEW, Array indexing has too few indices [LRM 4.1.1(3)]

Error.

INDXTOOMANY, Array indexing has too many indices [LRM 4.1.1(3)]

Error.

INITLIM, Initialization is not allowed for *entity* of limited type *entity* [LRM 7.4.4(6-8)]

Error.

INLINEASSUMED, Pragma INLINE assumed for *entity* [LRM 6.3.2; RTR 9.2.2]

Informational - compilation note.

INLINEDEPENDSES, Dependence established on *entity* for inline expansion of *entity* [LRM 10.3(7)]

Informational - compilation note.

INLINEEXPAND, Call of *entity* expanded inline [LRM 6.3.2; RTR 9.2]

Informational - compilation note.

INLINEEXPANDONL, Call of *entity* expanded inline because no code was generated for *entity* when its body was compiled

Informational - weak warning.

INLINEGENALSO, Pragma INLINE will also be ignored for previously compiled *entity*

Informational - supplemental.

INLINEGENASSUME, Pragma INLINE_GENERIC assumed for *entity* [LRM 12.1a; RTR 9.3.1]

Informational - compilation note.

INLINEGENBODY, *Entity* is not inlinable because of some construct within its body; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLINEINSTBODY, *Entity* from *entity* is not inlinable because of some construct within its body; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLINEINSTPROFI, *Entity* from *entity* is not inlinable because of its parameter or result profile; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLINEINSTTEMPL, *Entity* from *entity* is not inlinable because its generic body is not available; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLINENOEXPANDC, Call of *entity* is not expanded inline because *entity* is a containing subprogram [LRM 6.3.2; RTR 9.2.1]

Informational - compilation note.

INLINENOEXPANDN, Call of *entity* is not expanded inline because the generic body is not available or because the subprogram is not inlinable [LRM 6.3.2; RTR 9.2.1]

Informational - compilation note.

INLINENOEXPANDR, Call of *entity* is not expanded inline because it occurs within an inline expansion of itself [LRM 6.3.2; RTR 9.2.1]

Informational - compilation note.

INLINEONLY, Code generation suppressed for *entity*, which is always expanded inline [LRM 6.3.2; RTR 9.2.1]

Informational - compilation note.

INLINESUBPBODY, *Entity* is not inlinable because of some construct within its body; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLINESUBPSTUB, *Entity* is not inlinable because its subunit is not available; pragma *pragma_name* ignored in this unit [LRM 6.3.2; RTR 9.2.1]

Informational - weak warning.

INLINESUBPTEMPL, *Entity* is not inlinable because the generic body in which its body is contained is not available; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLSUBPROF, *Entity* is not inlinable because of its parameter or result profile; pragma *pragma_name* ignored [LRM 6.3.2; RTR 9.2.1]

Warning.

INLSUBPROFPRMT, *Entity* of *entity* is, or has subcomponents of, a task type

Informational - supplemental.

INLSUBPROFPRMU, The type of *entity* is unknown due to a previous error

Informational - supplemental.

INLSUBPROFRECO, *Entity* of function result is not constrained

Informational - supplemental.

INLSUBPROFRESTK, *Entity* of function result is, or has subcomponents of, a task type

Informational - supplemental.

INLSUBPROFRESUN, The type of the function result is not known due to a previous error

Informational - supplemental.

INOUTEXP, An expression is not allowed as an actual parameter corresponding to *entity* [LRM 6.4.1(3)]

Error.

INOUTTYPECONV, Operand of type conversion to *type_or_subtype* is itself also a type conversion (to *type_or_subtype*); only a variable name is allowed

Informational - supplemental.

INSBEFORE, Inserted *lexical_element* before *lexical_element*

Error.

INSERTNULLE, Inserted "null;" before "end"

Error.

INSERTNULLW, Inserted "null;" before "when"

Error.

INSINCMPLPARUN, Instantiation incomplete because the body for *entity*, which contains the generic body for *entity*, is not available

Informational - status.

INSINCMPLTMPSUN, Instantiation incomplete because subunit *entity*, which is, or is part of, the generic body for *entity*, is not available

Informational - status.

INSINCMPLTMPUN, Instantiation incomplete because the generic body for *entity* is not available

Informational - status.

INSMATCH, Inserted *lexical_element* to match *lexical_element source_location*

Error.

INSMATCHE, Inserted *lexical_element* to match *lexical_element* inserted *source_location*

Error.

INSSEMI, Inserted ";" at end of line

Error.

INSTASSUMENOPAC, *Entity* is used in *entity* without a generic package body

Informational - compilation note.

INSTASSUNCONSTR, *Entity* is used in *entity* with unconstrained actual *entity* corresponding to generic *entity*

Informational - supplemental.

INSTASSUNCSTRND, *Entity* requires a constrained subtype as the actual corresponding to generic *entity* [LRM 12.3.2(4), 13.7a.1, 13.10.2(2+)]

Error.

INSTASSUNCSTRWD, *Entity* requires a constrained subtype or a type with discriminants with defaults as the actual corresponding to generic *entity* [LRM 12.3.2(4), AI-00037]

Error.

INSTEXPANDINLIN, *Entity* expanded inline [LRM 12.1a, RTR 9.3.1]

Informational - compilation note.

INSTINCOMPLPARS, Instantiation incomplete because subunit *entity*, which contains the generic body for *entity*, is not available

Informational - status.

INSTINCOMPLPREV, Instantiation incomplete because the generic body for *entity* is not available or is incomplete (for the reason given in a prior message)

Informational - status.

INSTINCOMPLTEMP, Instantiation incomplete because the body for *entity*, which contains the generic body for *entity*, is not available

Informational - status.

INSTNOTEXPANDIN, *Entity* not expanded inline as the generic body for *entity* is not available [LRM 12.1a, RTR 9.3.1]

Informational - compilation note.

INTASK, *Entity* must be within a task body [LRM 9.9(5-6)]

Error.

INTASKINNERUNIT, *Entity* is not allowed within inner *entity* of *entity* [LRM 9.5(8)]

Error.

INTDEFNUL, Integer type definition has null range of *number .. number*

Informational - weak warning.

INVSTORSIZEPRE, *Entity* is not an access (sub)type, task type, or task object [LRM 13.2(7-10)]

Error.

LASTBIT_INVALID, Not a valid last bit position (too large or less than first bit) [LRM 13.4(5), F.9.5]

Error.

LENATTRDES, The length attribute designator must be one of SIZE, STORAGE_SIZE, or SMALL [LRM 13.2]

Error.

LENDEPSGENFORM, A *name* length clause is not allowed for *entity*, which is, or depends on, a generic formal type [LRM 13.1(10+,14), F.4]

Error.

LENGTH_SMALL_GT, Specified small *value* is finer than the small *value* for the parent type *entity* [LRM 3.4(4), 3.5.6(3), AI-00099]

Error.

LENGTH_SMALL_TO, Specified small is larger than the delta for *entity* [LRM 13.2(12)]

Error.

LEXAPOST, Invalid apostrophe ('); possible unterminated character literal or string literal delimited by apostrophe (') instead of by quotation marks (")

Error.

LEXBASE2_16, Base must be between 2 and 16; base 16 assumed [LRM 2.4.2(1)]

Error.

LEXBASECOLON, Colon delimiter illegal in sharp-delimited based number [LRM 2.10(3)]

Error.

LEXBASEDEL, Expecting based digit or *sharp_or_colon_character*; unexpected characters ignored

Error.

LEXBASEDELMISS, Expecting based digit or *sharp_or_colon_character*; remainder of line ignored

Error.

LEXBASEINT, Base of based literal must be an integer [LRM 2.4.2(2)]

Error.

LEXBASESHARP, Sharp delimiter illegal in colon-delimited based number [LRM 2.10(3)]

Error.

LEXCNTLCHAR, Illegal control character (*character_value*) in character literal [LRM 2.5(1)]

Error.

LEXCNTLCHARSTR, Illegal control character (*character_value*) in character string [LRM 2.5(1)]

Error.

LEXDOLLAR, Deleted illegal dollar sign ("$") [LRM 2.3(2)]

Error.

LEXDOUBUND, Illegal double underline; deleted "_" [LRM 2.3(2)]

Error.

LEXDOUBUNDIGN, Illegal double underline; deleted "_" [LRM 2.4.1(2), 2.4.2(2)]

Error.

LEXEOLCHAR, Illegal control character (*character_value*) in character string; remainder of line ignored [LRM 2.5(1)]

Error.

LEXEXPINT, Exponent must be an integer [LRM 2.4.1(2)]

Error.

LEXIGNAFTERFE, Ignored characters in comment (beginning with *character_value*) that follow a format effector sequence beginning with *character_value* [LRM 2.2(3+)]

Error.

LEXIGNSPACE, Unexpected space ignored

Error.

LEXILLCHAR, Illegal character (*character_value*) ignored

Error.

LEXILLDIG, Illegal digit for base *number*; "0" assumed [LRM 2.4.2(4)]

Error.

LEXINSSPACE, Inserted missing space after number *number* [LRM 2.2(4)]

Error.

LEXINTEXP, Illegal minus sign in exponent of integer literal ignored [LRM 2.4.1(4)]

Error.

LEXLEADUND, Deleted illegal leading underline [LRM 2.3(2)]

Error.

LEXLEADUNDIGN, Deleted illegal leading underline [LRM 2.4.1(2), 2.4.2(2)]

Error.

LEXLEADZERO, Missing digit; inserted "0" before "." [LRM 2.4.1(2), 2.4.2(2)]

Error.

LEXLONGNUM, Number exceeds the implementation limit of *number* characters after correction; "0.0" assumed

Error.

LEXMISSBASED, Missing based number [LRM 2.4.2(1)]

Error.

LEXMISSDIGDOT, Missing digit; inserted "0" after "." [LRM 2.4.1(2), 2.4.2(2)]

Error.

LEXMISSEXP, Missing exponent [LRM 2.4.1(2)]

Error.

LEXQUOTE, Quotation character illegal in percent-delimited character string [LRM 2.10(4)]

Error.

LEXREPSPACE, Replaced unexpected space with "0" [LRM 2.4.2(1)]

Error.

LEXREPUND, Missing digit; replaced "_" with "0" [LRM 2.4.2(1)]

Error.

LEXTRAILUND, Deleted illegal trailing underline [LRM 2.3(2)]

Error.

LEXTRAILUNDIGN, Deleted illegal trailing underline [LRM 2.4.1(2), 2.4.2(2)]

Error.

LEXUNDIGN, Deleted illegal underline [LRM 2.4.1(2)]

Error.

LEXUNTERMSTR, Unterminated character string

Error.

LOGOPMIXED, Logical operators must not be intermixed (without separating parentheses) [LRM 4.4(2,6)]

Error.

MAXARRDIMS, Array type definition has *number* dimensions, which exceeds the implementation limit of *number* [LRM F.9.5]

Error.

MAXCALLPARAMS, Subprogram has *number* formal parameters, which exceeds the implementation limit of *number* [LRM F.9.5]

Error.

MAXENUMLITS, Enumeration type definition has *number* literals, which exceeds the implementation limit of *number* [LRM F.9.5]

Error.

MERGE, Merged *lexical_element* and *lexical_element* to form *lexical_element*

Error.

MISMATCON, Constraints on *entity* do not match those on *entity*

Informational - weak warning.

MULLOOPBLOCK, Found identifier when expecting one of { "declare" "begin" "for" "while" "loop" }

Error.

MUST_HAVE_OTHER, Must have an 'others' choice [LRM 3.7.3(4), 4.3(5), 5.4(5)]

Error.

NAMENOTVALUED, Name is not a kind of entity that has a value

Error.

NAMENOTVALUED1, *Entity* is not a kind of entity that has a value

Error.

NAMENOTVAR, Name is not a variable [LRM 5.2(1)]

Error.

NAMENOTVAR1, *Entity* is not a variable [LRM 5.2(1)]

Error.

NAMERESTYPUNK, Result type of name is unknown due to a prior error

Error.

NAMERESTYPUNK1, Result type of *entity* is unknown due to a prior error

Error.

NOBODY, *Entity* has no corresponding body [LRM 6.3(3), 7.1(4), 9.1(1), 12.2(2)]

Error.

NOBODYIMPT, *Entity* has neither a corresponding body nor an INTERFACE pragma [LRM 6.3(3), 13.9(3)]

Error.

NOCODEGEN, This code is not reachable

Informational - compilation note.

NODISCRIM, *Entity* for *entity* does not have discriminants [LRM 3.7.4(2-3)]

Error.

NOENTRYFAMINDX, Entry index for entry family *entity* is missing [LRM 9.5(2)]

Error.

NOFULLCONST, Deferred *entity* has no corresponding full constant [LRM 7.4.3(1)]

Error.

NONCTCUPL, Size of aggregate exceeds the implementation limit of MAX_INT bytes

Error.

NONUNILAB, *Entity* is not unique as a label, or block or loop name in this body; it also occurs *source_location* [LRM 5.1(4)]

Error.

NOPACKSPEC, No package specification occurs previously for *entity* [LRM 7.1(1)]

Error.

NOSELECTED, No selected component named *name* is defined for any of the possible meanings of the prefix [LRM 4.1.3]

Error.

NOSELECTED1, No selected component named *name* is defined for *entity* [LRM 4.1.3]

Error.

NOSELECTED1RES, No selected component named *name* is defined for *entity* with result *entity* [LRM 4.1.3]

Error.

NOSELECTED1UNK, No selected component named *name* is defined for *entity* (with result type unknown due to a prior error) [LRM 4.1.3]

Error.

NOSELECTEDRENA, *Entity* is not allowed as a prefix within the renamed *entity* [LRM 4.1.3]

Error.

NOSRCINFOANA, No source file information will be included in the analysis data file for the recompilation of *kind name*; original source file was compiled with the /NONOTE_SOURCE qualifier

Informational - weak warning.

NOSRCINFODIA, No source file information will be included in the diagnostic file for the recompilation from copied source file *file_specification*; original source file was compiled with the /NONOTE_SOURCE qualifier

Informational - weak warning.

NOTACCTASKTYPE, *Entity* is neither an access nor a task type or subtype

Error.

NOTACCTYPE, *Entity* is not an access type or subtype

Error.

NOTALLOCATED, Variable *variable_name* is not referenced; therefore, it is not allocated

Informational - compilation note.

NOTARRDISCTYPE, *Entity* is neither an array type nor a type with discriminants

Error.

NOTARRTYPE, *Entity* is not an array type or subtype

Error.

NOTARRTYPEONEDI, *Entity* has more than one dimension

Error.

NOTARRTYPOBJ, *Entity* is not an array object or value, nor an array type or subtype, nor an access value that designates an array object

Error.

NOTASKSPEC, No task specification occurs previously for *entity* [LRM 9(4)]

Error.

NOTCONSTR, *Entity* is not constrained [LRM 3.2.1(1), 4.8(4)]

Error.

NOTDECL, *Name* is not declared [LRM 8.3]

Error.

NOTDECLALLHIDER, All declarations with designator *designator* are globally hidden by *entity* [LRM 8.3(16)]

Error.

NOTDECLALLHIDIN, All declarations with designator *designator* are globally hidden by *entity* [LRM 8.3(16)]

Informational - supplemental.

NOTDECLMUTUAL, Potentially visible declarations from use clauses are not visible because of mutual hiding, including (at least) *entity* [LRM 8.4(6)]

Informational - supplemental.

NOTDECLOPOPND, *Entity* is not declared for the given operands [LRM 4.5]

Error.

NOTDFLTINTFORM, Default resolution to the type INTEGER does not apply because one or both expressions is not a literal, named number, or attribute; however, the type INTEGER is assumed [LRM 3.6.1(2)]

Informational - supplemental.

NOTDISCRIM, *Entity* is not a discriminant of the containing record definition [LRM 3.7.3(1)]

Error.

NOTDISCTYPE, *Entity* is not a discrete type or subtype

Error.

NOTENCLOOP, *Entity* is not for a loop that encloses this exit statement [LRM 5.7(3)]

Error.

NOTENTRY, *Entity* is not an entry or entry family

Error.

NOTENUMTYPE, *Entity* is not an enumeration type or subtype

Error.

NOTEQOP, *Entity* is not an equality operator [LRM 6.7(5)]

Error.

NOTERENA, *Entity* is renamed as *entity*

Informational - supplemental.

NOTERENAP, *Entity* is renamed in part as *entity*

Informational - supplemental.

NOTEVALNONSTATI, Entity is not evaluated due to a prior error; assume nonstatic

Error.

NOTFIXEDTYPE, *Entity* is not a fixed-point type or subtype

Error.

NOTFLOATTYPE, *Entity* is not a floating-point type or subtype

Error.

NOTINDXCALLABLE, *Entity* is not a kind of entity that can be indexed or called as a function

Error.

NOTINLOOP, Exit statement is not enclosed in a loop statement [LRM 5.7(1)]

Error.

NOTPACKSPEC, *Entity* is not a package specification [LRM 7.1(3)]

Error.

NOTPRIVNOW, Attribute CONSTRAINED is illegal after the full declaration of *entity* [LRM 7.4.2(9-10)]

Error.

NOTPRIVTYPE, *Entity* is not a private type or subtype

Error.

NOTREALTYPE, *Entity* is not a real type or subtype

Error.

NOTRECCOMP, *Entity* of *entity* is not a component of a record [LRM 13.7.2(7-10)]

Error.

NOTRECCOMPOF, *Entity* is not a component of record type *entity* [LRM 13.4(5)]

Error.

NOTRECTYPE, *Entity* is not a record type or subtype

Error.

NOTREFCOPYBACK, *Entity* of *type* is not referenced within *entity*; in particular, no value is ever assigned to be copied back to the actual parameter [LRM 6.2(4-6)]

Informational - weak warning.

NOTREFCOPYBACKF, *Entity* of *type*, with corresponding full *type*, is not referenced within *entity*; in particular, no value is ever assigned to be copied back to the actual parameter [LRM 6.2(4-6)]

Informational - weak warning.

NOTREFWITHIN, *Entity* is not referenced within *entity*

Informational - compilation note.

NOTSAMEPARTSPEC, *Entity* is not declared in this declarative part, package specification, or task specification [LRM 13.1(5)]

Error.

NOTSAMEPROFIL, *Entity* does not have the same parameter and result type profile as *entity* [LRM 6.6]

Error.

NOTSCALTYPE, *Entity* is not a scalar type or subtype

Error.

NOTSCAORARR, *Entity* is not a scalar type or subtype, nor an array object or value, nor an array type or subtype, nor an access value that designates an array object [LRM 3.5(7-9), 3.6.2(2-10)]

Error.

NOTSIMPLEPREFIX, The prefix of the attribute that appears in a length clause must be a simple name [LRM 13.2, AI-00300]

Error.

NOTSTMTCALLABLE, *Entity* is not a kind of entity that is callable as an entry or procedure [LRM 6.4(1), 9.5(6-7)]

Error.

NOTTASKOBJ, *Entity* is of *entity*, which is not a task type [LRM 9.9(2-3)]

Error.

NOTTASKTYPE, *Entity* is not a task type [LRM 9.1(1)]

Error.

NOTTASKTYPEOBJ, *Entity* is not a task type or single task (task object) [LRM 9.1(1)]

Error.

NOTTYPORSUB, *Entity* is not a type or subtype

Error.

NOTVALUED, *Entity* is not a kind of entity that has a value

Error.

NOTYETDEN, *Entity* is not yet denotable [LRM 8.3(5,22)]

Error.

NOT_A_PASSABLE, *Entity* cannot be passed by an A or NCA descriptor; it is not a byte-aligned array of 1-bit or byte-aligned components [LRM 13.9a.1.2]

Warning.

NOT_DESC_PASSAB, *Entity* cannot be passed by any form of DESCRIPTOR mechanism [LRM 13.9a.1.2]

Warning.

NOT_EXACT_DELTA, The specified small is not an exact power of 2 [LRM 13.2(12+)]

Error.

NOT_EXHAUSTIVE, Missing values in the range of *entity* [LRM 3.7.3(3), 4.3(5), 5.4(4)]

Error.

NOT_EXHAUSTIVE_, Missing values in range *low* .. *high* of *entity* [LRM 4.3(6)]

Error.

NOT_REFERENCE_P, *Entity* cannot be passed by the REFERENCE mechanism [LRM 13.9a.1.2]

Warning.

NOT_S_PASSABLE, *Entity* cannot be passed by an S or SB descriptor; it is not a scalar type, access type, nor a one-dimensional array of unsigned byte components [LRM 13.9a.1.2]

Warning.

NOT_UBA_PASSABL, *Entity* cannot be passed by a UBA descriptor; it is not an array [LRM 13.9a.1.2]

Warning.

NOT_UBS_PASSABL, *Entity* cannot be passed by a UBS or UBSB descriptor; it is not a one-dimensional array of 1-bit components [LRM 13.9a.1.2]

Warning.

NOT_VALUE_PASSA, *Entity* cannot be passed by the VALUE mechanism [LRM 13.9a.1.2]

Warning.

NO_ADEQUATE_PRE, No predefined type is able to satisfy these requirements [LRM 3.5.7, 3.5.9, F.9.3]

Error.

NULLCOMPALONE, A null component must occur alone in a component list [LRM 3.7(2)]

Error.

NULL_RANGE_ILLE, Null range must be the only choice [LRM 4.3.2(3)]

Error.

NUMERICNOCONSTR, When a handler for NUMERIC_ERROR is provided, it should also include a choice for CONSTRAINT_ERROR [AI-00387]

Warning.

NUMOPUNARY, The operand following a binary adding, multiplying, or exponentiation operator must not begin with a unary adding operator (without separating parentheses) [LRM 4.4(2,6), 4.5(2)]

Error.

OBJECTARRDEFCON, An array type definition that occurs in an object declaration must be constrained [LRM 3.2(9)]

Error.

OBJECTATTRRES, The result of attribute *attribute_name* is a value, not an object [LRM 3.2.1, A]

Error.

OBJECTFUNCRES, The result of calling *entity* is a value, not an object [LRM 3.2.1, 6.5(1)]

Error.

OBJECTOPRES, The result of operator *entity* is a value, not an object [LRM 3.2.1, 4.4(1)]

Error.

OPDECLDFLT, A default expression is not allowed for a parameter of an operator declaration [LRM 6.7(2)]

Error.

OPDECLNG, A declaration for operator *designator* is not allowed [LRM 6.7(1)]

Error.

OPDECLONE, An operator declaration for *designator* must have one parameter [LRM 6.7(2)]

Error.

OPDECLONETWO, An operator declaration for *designator* must have either one or two parameters [LRM 6.7(2)]

Error.

OPDECLTWO, An operator declaration for *designator* must have two parameters [LRM 6.7(2)]

Error.

OPEQPARAMLIM, *Entity* of parameter in an equality operator declaration is not limited [LRM 6.7(4)]

Error.

OPEQRESBOOL, Result type *entity* is not the predefined type STANDARD.BOOLEAN [LRM 6.7(4)]

Error.

OPEQSAME, Type *entity* of *entity* is not the same as type *entity* of *entity* [LRM 6.7(4)]

Error.

OPGENUNIT, An operator designator is not allowed as the name of a generic function declaration [LRM 12.1(4)]

Error.

OPLIBUNIT, An operator designator is not allowed as the name of a library unit or subunit [LRM 10.1(3)]

Error.

OPTIM_SUPPRESSE, Subprogram too large for normal optimization; compilation continues with some optimization suppressed

Warning.

OTHSLAST, An 'others' choice must be the last choice [LRM 3.7.3(4), 4.3(5), 5.4(5), 11.2(5)]

Error.

OTHSNG, An 'others' choice is not allowed in this context [LRM 4.3.2(4-8)]

Error.

OTHSONLY, An 'others' choice must be the only choice [LRM 3.7.3(4), 4.3(5), 5.4(5), 11.2(5)]

Error.

OUTLIMNOTPACK, 'Out' parameter is in a subprogram or entry that is not declared in the visible declarations of the same package as limited type *entity* [LRM 7.4.4(4)]

Error.

OUTLIMNOTPRIV, Type *entity* of 'out' parameter is of a limited, but not a private, type [LRM 7.4.4(4)]

Error.

OUTSIDE_RANGE, *Number_or_enumeral* is outside the range of allowable values, *number_or_enumeral .. number_or_enumeral* [LRM 3.7.3(3), 4.3(5), 5.4(4)]

Error.

PARASSFORMID, Only an identifier is allowed as a choice in a named association of a subprogram call [LRM 6.4(2,3)]

Error.

PARASSONECHO, Only one choice is allowed in a named association of a subprogram call or generic instantiation [LRM 6.4(2,3), 12.3(2,3)]

Error.

PARDECLACC, Illegal parameter declaration; found access type definition when expecting a type or subtype name [LRM 6.1(2)]

Error.

PARDECLARRAY, Illegal parameter declaration; found array type definition when expecting a type or subtype name [LRM 6.1(2)]

Error.

PARDECLREC, Illegal parameter declaration; found record type definition when expecting a type or subtype name [LRM 6.1(2)]

Error.

PARSESTACK, Terminating compilation because syntax nesting level is too deep

Fatal.

PARTYPCHODFLT, The parent type chosen for *name* is the predefined type LONG_FLOAT (D_floating representation)

Informational - compilation note.

PARTYPCHOFFLT, The parent type chosen for *name* is the predefined type FLOAT (F_floating representation)

Informational - compilation note.

PARTYPCHOGFLT, The parent type chosen for *name* is the predefined type LONG_FLOAT (G_floating representation)

Informational - compilation note.

PARTYPCHOHFLT, The parent type chosen for *name* is the predefined type LONG_LONG_FLOAT (H_floating representation)

Informational - compilation note.

PARTYPCHOINT, The parent type chosen for *name* is the predefined type INTEGER (signed longword representation)

Informational - compilation note.

PARTYPCHOSINT, The parent type chosen for *name* is the predefined type SHORT_INTEGER (signed word representation)

Informational - compilation note.

PARTYPCHOSSINT, The parent type chosen for *name* is the predefined type SHORT_SHORT_INTEGER (signed byte representation)

Informational - compilation note.

PASS_MECH_IS, Selected or specified passing mechanism is *mechanism_name*

Informational - compilation note.

PM_NAMDOENOTMAT, Name *name* does not match *entity source_location* [LRM 5.5(3), 5.6(3), 6.3(4), 7.1(3), 9.5(7)]

Error.

PM_NOENDID, Name *name* not specified at end of *block_or_loop* starting *source_location* [LRM 5.5(3), 5.6(3)]

Error.

PM_NOSTARTID, Name *name* not specified at start of *block_or_loop source_location* [LRM 5.5(3), 5.6(3)]

Error.

POSASSAFTNAM, A positional association must not occur after a named association [LRM 4.3(4), 6.4(4), 12.3(3)]

Error.

POSSONECOMPAGGR, Possibly an aggregate of *entity* with one component is intended; this requires use of named notation for the component [LRM 4.3(4)]

Informational - supplemental.

POSSSUBTYPE, Possibly a subtype declaration is intended

Informational - supplemental.

POSSUSECALL, Possibly a selected component of (or use clause for) package *entity* is intended; this would make the following visible for legal calls

Informational - supplemental.

POSSUSENAME, Possibly a selected component of (or use clause for) package *entity* is intended; this would make the following visible

Informational - supplemental.

PRAGINPRAG, Illegal pragma appears as a pragma argument; pragma ignored

Error.

PRIORADRREP, An address representation clause is already given *source_location* for *entity* [LRM 13.1(4)]

Error.

PRIORASSOC, An association for *entity* is already given in this parameter list [LRM 4.3(6), 6.4(5), 12.3(3)]

Error.

PRIORCOMPREP, A component representation is already given for *entity* in this record representation clause [LRM 13.4(6)]

Error.

PRIORCONSTR, *Entity* is already constrained *source_location* [LRM 3.6.1(3), 3.7.2]

Error.

PRIORRECREP, A record representation clause is already given *source_location* for *entity* [LRM 13.1(3)]

Error.

PRIORSIZE, A SIZE representation specification is already given *source_location* for *entity* [LRM 13.1(3)]

Error.

PRIORSMALL, A SMALL representation specification is already given *source_location* for *entity* [LRM 13.1(3)]

Error.

PRIORSTORSIZE, A STORAGE_SIZE representation specification is already given *source_location* for *entity* [LRM 13.1(3)]

Error.

PRIORSTUBSAMENA, A stub declaration named *name* is already given *source_location* [LRM 10.2(5)]

Error.

PRIORSTUBSUPPL, The simple names of all subunits that have the same ancestor library unit must be distinct

Informational - supplemental.

PRIORTERMALT, A terminate alternative is already given *source_location* [LRM 9.7.1(3)]

Error.

PRIVNOFULLTYPE, *Entity* has no corresponding full type declaration [LRM 7.4.1(1)]

Error.

PRIVTYPEPACKGEN, A private type declaration is allowed only in the visible part of a package specification or a generic formal part [LRM 7.4(3)]

Error.

RAIWONAM, A raise statement without an exception name is allowed only in the exception part of block or body [LRM 11.3(3)]

Error.

RANASSUB, Attribute RANGE is not allowed as a subtype indication [LRM 3.3.2(2), 3.5(2)]

Error.

RANGENOTSTATIC, Range is not static [LRM 4.9(11)]

Error.

RANGENOTSTATICE, Apparent static range raises an exception during evaluation; assume nonstatic [LRM 4.9(2)]

Error.

RANGEUNIVINT, Type {universal_integer} is not allowed for the discrete range of a constrained array definition, an iteration rule, or an index of an entry family [LRM 3.6.1(2)]

Error.

READNOTALLOW, Reading from *entity* is not allowed [LRM 6.2(5)]

Error.

RECAGGRBADVAR, Resolution is not complete because of a prior error related to the variant part *source_location*

Informational - supplemental.

RECAGGRNOCHOICE, Value corresponding to *entity* does not match any choice of the variant part *source_location* [LRM 4.3.1(2)]

Error.

RECAGGRTYPE, Error occurs in record aggregate for *entity*

Informational - supplemental.

RECREPDERIVED, A record representation clause is not allowed for derived *entity* whose parent *entity* has derivable subprograms (such as *entity*) [LRM 13.1(3), 13.4]

Error.

REC_ALIGN_BAD, Record alignment *number* is not a power of 2 between 1 and 512 [LRM 13.4(4+)]

Error.

REC_ALIGN_LEQ_Z, Record alignment must be greater than zero [LRM 13.4(4+)]

Error.

REC_ALIGN_TOO_B, Record alignment is too large [LRM 13.4(4+)]

Error.

REC_TOO_LARGE, CONSTRAINT_ERROR or NUMERIC_ERROR will be raised if the size of this (sub)type is computed [LRM 11.1(6), F.9.5, AI-00387]

Informational - weak warning.

REDECLDERIV, *Entity* hides *entity* within its immediate scope [LRM 8.3(17)]

Informational - compilation note.

REDECLDERIVHIDD, *Entity*, derived from *entity*, is hidden by *entity* [LRM 8.3(17)]

Informational - compilation note.

REDECLDERIVTYPE, *Entity*, derived from *entity*, illegally redeclares *entity*, derived from *entity* [LRM 3.4(11), 8.3(17)]

Error.

REDECLENUM, *Entity* hides derived *entity* within its immediate scope [LRM 8.3(17)]

Informational - compilation note.

REDECLENUMHIDDE, Derived *entity* is hidden by *entity* [LRM 8.3(17)]

Informational - compilation note.

REDECLERR, Illegal redeclaration of *entity* [LRM 8.3(17)]

Error.

REDECLIMPL, *Entity* hides *entity* within its immediate scope [LRM 8.3(17)]

Informational - compilation note.

REDECLLIB, Illegal redeclaration of library *entity* [LRM 8.3(17), 10.1(3-5)]

Error.

REDECLSAME, Illegal redeclaration of *entity* given earlier in this type declaration [LRM 3.5.1(3)]

Error.

REDECLSECLIB, Illegal secondary *unit_kind* corresponding to library *entity* [LRM 10.1(3-5)]

Error.

REDECLSECLIBPRO, Illegal secondary *unit_kind* corresponding to library *entity*; the specification and body do not have the same parameter and result type profile [LRM 6.6(1), 10.1(3,6)]

Error.

RELOPMIXED, Relational operators must not be intermixed (without separating parentheses) [LRM 4.4(2,6)]

Error.

RENAATTRRES, The result of attribute *attribute_name* cannot be renamed as an object [LRM 8.5(4)]

Error.

RENAFUNCRES, The result of calling *entity* cannot be renamed as an object [LRM 8.5(4)]

Error.

RENAOBJRENA, *Entity* cannot be renamed as an object [LRM 8.5(4)]

Error.

RENAOPRES, The result of operator *entity* cannot be renamed as an object [LRM 8.5(4)]

Error.

REPABBREV, Replaced abbreviation *lexical_element* with *lexical_element*

Error.

REPFORCED, The representation of *entity* has already been forced *source_location* [LRM 13.1(6)]

Error.

REPFORCEDHERE, The representation of *entity* is forced here

Informational - supplemental.

REPLACE, Replaced *lexical_element* with *lexical_element*

Error.

RESTYPEXPUNKN, Result type of expression is unknown due to a prior error

Informational - supplemental.

RESTYPINCON, Result type of expression is inconsistent with its context, which requires *type_or_subtype* [LRM 8.7]

Error.

RESTYPNAMUNKN, Result type is unknown due to a prior error related to *entity*

Informational - supplemental.

RETEXPPROCACC, A return statement with an expression is not allowed in a procedure body or an accept statement [LRM 5.8(4)]

Error.

RETEXPREQ, A return expression is required within a function [LRM 5.8(4), 6.5(1)]

Error.

RETNOTALL, A return statement is not allowed in this context [LRM 5.8(3)]

Error.

RSCHINCOPRIERR, Resolution or checking of *name* is incomplete because of a prior error related to *entity*

Error.

RSLINCOPRIORERR, Resolution of *name* is incomplete because of a prior error related to *entity*

Error.

SELECTDELAYTERM, Selective wait statement has both a delay alternative *source_location* and a terminate alternative *source_location* [LRM 9.7.1(3)]

Error.

SELECTELSEDELAY, Selective wait statement has both an else part *source_location* and a delay alternative *source_location* [LRM 9.7.1(3)]

Error.

SELECTELSETERM, Selective wait statement has both an else part *source_location* and a terminate alternative *source_location* [LRM 9.7.1(3)]

Error.

SELECTNOACPT, Selective wait statement has no accept alternative [LRM 9.7.1(3)]

Error.

SELREPSMA, Selected representation is *discrete representation*; selected small is 2**(*number*)

Informational - compilation note.

SEPNG, A separate clause is allowed only with a subprogram or package body or a task body, not *a_kind* [LRM 10.2(2)]

Error.

SHAREGENASSUMED, Pragma SHARE_GENERIC assumed for *entity*

Informational - compilation note.

SHAREGENBODYCRE, A new shareable body has been created for *entity*, for use with *entity*

Informational - compilation note.

SHAREGENBODYUSE, An existing shareable body for *entity* has been used for *entity*

Informational - compilation note.

SHAREGENSPECCRE, A new shareable specification has been created for *entity*, for use with *entity*

Informational - compilation note.

SHAREGENSPECUSE, An existing shareable specification for *entity* has been used for *entity*

Informational - compilation note.

SHAREGENSUBCREA, A new shareable body has been created for *entity* within *entity*, for use with *entity*

Informational - compilation note.

SHAREGENSUBUSE, An existing shareable body for *entity* within *entity* has been used for *entity*

Informational - compilation note.

SIZE_NOT_BETWEE, The number of bits to represent *entity* must be between *number* and *number* [LRM 13.2(6+), 13.4(5+)]

Error.

SIZE_NOT_CONSTA, Size specification is not allowed because the size of *entity* is not constant [LRM 13.2(6)]

Error.

SIZE_NOT_EXACT, The number of bits to represent *entity* must be exactly *number* [LRM 13.2(6+), 13.4(5+)]

Error.

SLICEPREV, Error occurs in slice of *entity* of *type*

Informational - supplemental.

SOMDISCDEFLT, Some, but not all, discriminants have default expressions [LRM 3.7.1(4)]

Error.

SPECORREP, Replaced *lexical_element* with *lexical_element*

Error.

SRC_LINE_TOO_LO, Source line is *number* characters longer than the implementation limit of *number*—excess characters ignored

Error.

STATIC_ALIGN, The maximum alignment for static objects is 512 (page) [LRM 13.4(4+)]

Warning.

STMTINDECL, Ignored statement appearing within a list of declarations

Error.

STMTINPRAG, Illegal statement appears as a pragma argument; pragma ignored

Error.

STORAGE_ERROR, STORAGE_ERROR will be raised here [LRM 11.1(8)]

Informational - weak warning.

STORAGE_UNIT_IN, Not a valid storage unit (too large or negative) [LRM 13.4(5+)]

Error.

STORSIZEDERACC, A STORAGE_SIZE representation specification is not allowed for *entity*, a type derived from an access type [LRM 13.2(8)]

Error.

STRING_TOO_LONG, Value of the predefined type STRING, or a type derived from STRING, exceeds 65535 characters [LRM 3.6.3(1+), F.9.5]

Informational - weak warning.

STRNOTOP, String *string_literal* is not an operator designator [LRM 4.5(2)]

Error.

STUBPOS, A body stub declaration is allowed only in the outermost declarations of a library unit or subunit body [LRM 10.2(3)]

Error.

SUBPACTPREV, Error occurs in actual corresponding to *entity*

Informational - supplemental.

SUBPRENAPREV, Error occurs in default expression of *entity* corresponding to *entity*

Informational - supplemental.

SUBPRENAPREVI, Error occurs in default expression of *entity* corresponding to implicit formal of *entity*

Informational - supplemental.

SUBTNOTSTATIC, *Expression_range_or_subtype* is not static [LRM 4.9]

Error.

SUBTYPENULL, This is a null range

Informational - compilation note.

SUPP*, *Possible_meaning*

Informational - supplemental. Following detection of an error in overload resolution for a choice, expression, name, or subtype indication, a series of supplementary messages report the possible meanings that were considered for each component, presented in left-to-right order. There are multiple variations of this message, depending on the nature of the component (identifier, parentheses, attribute, and so on), on the number of possible meanings (none, one, or multiple), and on the kind of each meaning (valued entity, nonvalued entity, call of user function or predefined operator, and so on). A possible meaning that was determined to be inconsistent with its use in context is noted as "(discarded)" at the end of its description.

TERMCONTEXT, Compilation of *entity* terminated due to errors in the context clause

Error.

TERMHERE, Ada compilation terminated here

Informational - weak warning.

TERMHERE2, Compilation terminated here (secondary processing location)

Informational - weak warning.

TERMSUBUNIT, Compilation of *entity* terminated due to errors in the name specified for the parent unit of *entity*

Error.

TERMSUBUNITCONT, Compilation of *entity* terminated due to errors in the context clause and in the name specified for the parent unit of *entity*

Error.

TERMSYNTAX, Terminating compilation due to syntax errors in file *file_specification*

Error.

TERMUSER, Ada compilation terminated by user

Informational - weak warning.

TERMUSER2, Compilation terminated by user (secondary processing location)

Informational - weak warning.

TOOMANYLINES, Number of lines in source file exceeds implementation limit of *number*; terminating compilation

Fatal.

TOO_MANY_DISCRI, The number of discriminants exceeds the implementation limit of *number* [LRM F.9.5]

Error.

TOO_MANY_PSECTS, There are more than the implementation limit of *number* program sections; pragma PSECT_OBJECT ignored [LRM F.9.5]

Warning.

TOO_MANY_UNCONS, The number of unconstrained record parameters exceeds the implementation limit of *number* [LRM F.9.5]

Error.

TYPECONTEXTUNKN, Type checking is not complete; the type required from context is unknown due to a prior error

Informational - supplemental.

TYPECONVARRCOMC, Component type *entity* for array type *entity* is constrained *source_location*, but component type *entity* for *entity* is not constrained [LRM 4.6(11)]

Error.

TYPECONVARRCOMT, Component type *entity* of operand of array type *entity* is not the same as component type *entity* of target type *entity* [LRM 4.6(11)]

Error.

TYPECONVARRDIMS, Array type *entity* of operand has *number* dimensions, but target array type *entity* has *number* dimensions [LRM 4.6(11)]

Error.

TYPECONVARRIND, Type *entity* of index *number* of operand type *entity* is not convertible to index type *entity* of target type *entity* [LRM 4.6(11)]

Error.

TYPECONVFORM, A type conversion does not allow a null literal, an allocator, an aggregate, or a string literal as the argument [LRM 4.6(3)]

Error.

TYPECONVNOTALL, Type conversion to *entity* from operand result *entity* is not allowed [LRM 4.6]

Error.

TYPECONVNUMERIC, *Entity* is numeric, but result *entity* of operand is not numeric [LRM 4.6(7)]

Error.

TYPECONVONEARG, A type conversion must have only one argument [LRM 4.6(2)]

Error.

TYPEDERIVES, *Entity* is derived from *entity* [LRM 3.4(11)]

Informational - compilation note.

TYPERANGECONT, Type *entity* of range is not the same as type *entity* required from context [LRM 8.7]

Error.

TYPEWODISCRIMS, *Entity* has no discriminants [LRM 3.7.2(3)]

Error.

UNARYOPMIXED, A unary operator must not precede another unary operator (without separating parentheses) [LRM 4.4(2,6)]

Error.

UNCH_CONV_ACC_1, The representation of *access_type* is the address of a descriptor (because the designated *type* is unconstrained), while the representation of *access_type* is the address of the designated object

Informational - weak warning.

UNCH_CONV_ACC_2, The representations of *access_type* and *access_type* are the address of descriptors that do not have the same CLASS and DTYPE

Informational - weak warning.

UNCH_CONV_ACC_A, The representation of *access_type* is the address of a descriptor (because the designated *type* is unconstrained), not the address of the designated object

Informational - weak warning.

UNCH_CONV_SUSPE, This unchecked conversion (*entity*) is suspect

Informational - weak warning.

UNITINWITH, Library *entity* is named in a with clause *source_location*

Informational - supplemental.

UNITSOURCE, Library *entity* originated in source file *file_specification*

Informational - supplemental.

UNIVFIXCONV, Multiplication or division of values of *entity* must be explicitly converted to another numeric type [LRM 4.5.5(11)]

Error.

UNIVFIXUNIVOPND, Multiplication or division for *entity* is not allowed with an operand of type {universal_real} [LRM 4.5.5(10-11)]

Error.

UPDATEATTRRES, The result of attribute *attribute_name* cannot be updated [LRM 3.2.1, 4.1.4]

Error.

UPDATEDISCRIM, Update of *entity* is not allowed [LRM 3.2.1, 3.7.1(9)]

Error.

UPDATEFUNCRES, The result of calling *entity* cannot be updated [LRM 3.2.1, 6.5, 6.7]

Error.

UPDATENOTALLOW, Update of *entity* is not allowed [LRM 3.2.1(2), 6.2(1,3)]

Error.

UPDATEOPRES, The result of operator *entity* cannot be updated [LRM 3.2.1]

Error.

USEDEFCONST, Illegal use of deferred constant *entity* prior to its full constant declaration [LRM 7.4.3(2)]

Error.

USEINCO, Illegal use of *entity* prior to its full type declaration [LRM 3.8.1(4)]

Error.

USENOWITH, *Name* is not named in a prior with clause [LRM 10.1.1(3)]

Error.

USEPRIV, Illegal use of *entity* prior to its full type declaration [LRM 7.4.1(4)]

Error.

USEPRIVCOMP, Illegal use of *entity*, which has subcomponents of *type*, prior to the full declaration of that subcomponent type [LRM 7.4.1(4)]

Error.

USEPRIVCOMPANON, Illegal use of an anonymous array type, which has subcomponents of *type*, prior to the full declaration of that subcomponent type [LRM 7.4.1(4)]

Error.

USEWITHINSELF, A use of package *entity* while within that package has no effect [LRM 8.4(4)]

Informational - weak warning.

VALENUMREPRANG, Value *number* is outside the implemented range *number .. number* of enumeration representations [LRM F.9.5]

Error.

VALINTTYPERANG, Value *number* is outside the implemented range *number .. number* of any integer type [LRM F.9.5]

Error.

VALOUTIMPLRANG, Value *number* is outside the implemented range of *number .. number* (INTEGER'FIRST .. INTEGER'LAST) [LRM F.9.5]

Error.

VARPARTDISCRIMS, A variant part is allowed only in a record definition that has discriminants [LRM 3.7.3(3)]

Error.

VARPARTLAST, A variant part must occur last [LRM 3.7(2)]

Error.

WITHSELF, *Entity* depends on, but redefines, library *entity* (from source file *source_file*) [LRM 10.1(3-4)]

Error.

WRONGKIND, *Entity* is not *a_kind*

Error.

WRONGKINDAMBIG, Name is ambiguous; *a_kind* is required

Error.

WRONGKINDRES, Result of *entity* is not *type_or_subtype*

Error.

WRONGKINDWIINGE, Within a *generic_kind*, the generic acts as a nongeneric *kind* [LRM 12.1(5,10)]

Informational - supplemental.

WRONGKINDWIINTS, Within the body of a task, the task name refers to the task object that is currently executing the body; in particular, the name does not refer to the task type and cannot be used in a type mark [LRM 9.1(4)]

Informational - supplemental.

W_CONSTRAINT_ER, CONSTRAINT_ERROR would be raised here, but the qualifier /NOCHECK or pragma SUPPRESS_ALL is specified [LRM 11.1(5), 11.7(4+)]

Warning.

XFACENOIMPT, Pragma INTERFACE applies to multiple subprograms, but one or more import pragmas is missing [LRM 13.9(4+)]

Warning.

ZEROSIZEOBJ, Object *object_name* has zero size

Informational - compilation note.

Appendix G

ACS Diagnostic Messages

VAX Ada diagnostic messages generated by the program library manager are presented in this appendix. Some messages can occur for both the program library manager and the compiler; these are listed both in this appendix and in Appendix F.

The messages are listed in alphabetical order by ident. The ident is followed by the message text and the severity level of the message.

G.1 Diagnostic Message Format

The general format of a program library manager diagnostic message is as follows:

```
%Facility_code-Severity_code-Ident-Message_text
```

Facility_code
Is a four-letter code (ACS) that identifies the VAX Ada program library manager.

Severity_code
Is a letter (F, E, W, or I) that indicates the severity of the message. The meaning of these severity codes is discussed in Section G.2.

Ident
Is a name that uniquely identifies the message.

Message_text
Is a description of the event that has taken place. Italicized items in the message text in this appendix indicate items that are replaced with specific information when the message is generated.

G.2 Diagnostic Message Severity Codes

A VAX Ada program library manager diagnostic message contains one of four codes—F, E, W, I—which indicate the severity of the error:

- **F** indicates a fatal error. An F-level message indicates that the intended request cannot be executed.
- **E** indicates a user error. An E-level message is often supplemented with informational (I-level) messages that give additional information about the error.
- **W** indicates a warning message. A command may have performed some, but not all, of your request, and you may have to verify the command output.
- **I** indicates an informational message. An I-level message often contains supplementary information about a preceding or otherwise related E-level error.

G.3 ACS Diagnostic Messages

CL_ACCLIBDEN, Access to program library *directory_specification* is denied due to file protections associated with the VMS directory or the library index file

Error.

CL_ACUNOTFOU, *Entity* not found in *file_specification*

Error.

CL_BODALREXI, *Entity* already has a body

Error.

CL_BODNOTFOU, Body for *name* not found in library

Error.

CL_BODNOTFOU1, Body for *name* not found in library

Informational - weak warning.

CL_CHECKOK, All units current; no recompilations required

Informational.

CL_CLOSURE, The closure of the specified units is:

Informational.

CL_CMDERROR, Command error - *name*

Warning.

CL_CMDNOTSUPP, This command is not supported for this target

Warning.

CL_CNVTEDUNI, *Entity* is a converted unit and must be (re)compiled into the library

Error.

CL_COMIGNCTRLC, Command ignored due to CTRL/C

Warning.

CL_COMPILE, The following units need to be compiled from source files:

Informational.

CL_COMPILE1, The following units will be compiled from source files:

Informational.

CL_COMPILEOK, All units and files current; no compilations required

Informational.

CL_COMPILING, Invoking the VAX Ada compiler

Informational.

CL_COMPLETE, The following units need to be completed (use ACS COMPILE or ACS RECOMPILE):

Informational. Incomplete generic instantiations must be completed before you can link a program that contains such instantiations. An incomplete generic instantiation can occur if the body or subunits or the body for the corresponding generic unit are not available when the instantiation of the generic is compiled or if the generic body is compiled or recompiled after the unit containing the instantiation is compiled.

CL_COMPLETE1, The following units will be completed:

Informational.

CL_CONFLICT, Illegal combination of command elements; check documentation

Error.

CL_COPIED, *Entity* copied

Informational.

CL_DELDIRENT, Error deleting directory entry for *name*

Error.

CL_DELETED, *Entity* deleted

Informational - status.

CL_DELUNITS, *Name* deleted

Informational - status.

CL_DIRNOTEMP, Directory *file_specification_or_name* is not empty

Error.

CL_ENTERED, *Entity* entered

Informational.

CL_ERRACTCMS, Error activating shareable image CMSSHR

Error.

CL_ERRCREDIR, Error creating directory *file_specification*

Error.

CL_ERRCRELIB, Error creating program library *directory_specification*

Error.

CL_ERRDURCNV, Error converting *file_specification_or_name*; not converted

Error.

CL_ERRDURCOM, Error during compilation

Error.

CL_ERRDURCOP, Error copying *name*; not copied

Error.

CL_ERRDURENT, Error entering *name*; not entered

Error.

CL_ERRDURENT_CN, *Entity* not entered in converted library *file_specification*

Error.

CL_ERRDURLIN, Error during ACS LINK operation

Error.

CL_ERRDURLOA, Error loading *file_specification*

Error.

CL_ERRDURMER, Error during merge of *name*; not merged

Error.

CL_ERRFETCMS, Error fetching source for *entity*

Error.

CL_ERRINVSUB, Error invoking subprocess

Error.

CL_ERROPELIB1, Error opening program library *directory_specification*

Error. Either the library index file (ADALIB.ALB) or the library version control file (ADA$LIB.DAT) or both are not accessible or the VMS directory containing the program library is not specified correctly or not accessible

CL_ERRPRELOA, Error preloading *entity* in file *file_specification*

Error.

CL_ERRSEALIS, Error in search list specification *file_specification*

Error.

CL_ERRSUBMIT, Error submitting batch job

Error.

CL_EXCNETNOT, Exclusive access to compilation library *file_specification* across the network is not supported

Error.

CL_EXTRACTED, Source file for *entity* copied to *file_specification*

Informational.

CL_FILNOTREA, File *name* is not reachable from destination library

Error.

CL_FORCOPIED, *File_specification* copied as *entity*

Informational.

CL_FORENTERED, *File_specification* entered as *entity*

Informational.

CL_FORNOTFOU, *Entity* (foreign body) not found in *file_specification*

Error.

CL_ILLFMTDAT, The compilation unit file (.ACU) has illegal data format

Error.

CL_ILLFMTDIR, The library index file has illegal format

Error.

CL_ILLFMTHDR, The compilation unit file (.ACU) has illegal header format

Error.

CL_INCREFACU, Date-time mismatch on *file_specification* for *entity* in *directory_specification*

Error.

CL_INCREFFIL, Date-time mismatch for file *file_specification*

Error.

CL_INCREFFOR, Foreign body version mismatch on *file_specification* for *entity* in *directory_specification*

Error.

CL_INCREFUNI, *Entity* depends on *entity*, which has been redefined as *kind*

Error.

CL_INVACUFMT, The compilation unit file (.ACU) file has an invalid format

Error.

CL_INVLIBFMT, The library index file (ADALIB.ALB) has an invalid format

Error.

CL_INVPROCOD, Invalid protection code *protection_code*

Error.

CL_LIBALREXI, A program library already exists in directory *directory_specification*

Error.

CL_LIBCONACS, Library file specification *file_specification* must not contain an access control string

Error.

CL_LIBCRE, Library *file_specification* created

Informational.

CL_LIBCVT1, Version 1.n library *file_specification_or_name* converted in place to Version 2.0

Informational.

CL_LIBCVT2, Version 1.n library *file_specification_or_name* converted to Version 2.0 library *file_specification_or_name*

Informational.

CL_LIBCVTBEG2, Beginning conversion of library *file_specification* for Version 2.0

Informational.

CL_LIBCVTEND2, End of conversion of library *file_specification* for Version 2.0

Informational.

CL_LIBDEL, Library *file_specification* deleted

Informational.

CL_LIBDELABO, Deletion of *file_specification* aborted, library no longer consistent

Error.

CL_LIBIS, Current program library is *file_specification*

Informational.

CL_LIBNOTCVT2, Upgrade of library *name* for Version 2.0 was unsuccessful

Warning.

CL_LIBNOTDIR, *File_specification* is not a valid library directory

Error.

CL_LIBNOTSUB, *File_specification* is a library, not a sublibrary

Error.

CL_LINKING, Invoking the *name* for *name* target

Informational.

CL_LNKOPTCOP, Link option *name* copied

Informational.

CL_LNKOPTDEL, Link option *name* deleted

Informational - status.

CL_LOAD, The following files will be loaded:

Informational.

CL_LOADEDUNI, *Entity* is a loaded-only unit and must be (re)compiled into the library

Error.

CL_LOPALREXI, Link option *name* already exists in library; not replaced

Error.

CL_LOPCOPIED_V, An identical link option *name* exists in the current program library; no file transfer required; if actual file transfer is desired, first delete the link option and then copy

Informational.

CL_LOPNOTFOU, Link option *name* not found in library

Error.

CL_MAIUNIENT, Main unit *name* has no entry point

Error.

CL_MAIUNIKIN, Main unit *name* is *kind*, not a subprogram body

Error.

CL_MAIUNISIG, Main unit *name* has parameters or a result type that are not supported for a main program

Error.

CL_MAIUNISUB, Main unit *name* is a subunit, not a library subprogram

Error.

CL_MERGED, *Entity* merged

Informational.

CL_MISSDEP, The following units depend on missing units:

Informational.

CL_MISSSUBUNIT, The following units have missing subunits:

Informational.

CL_NAMETOOLONG, Name *name* is too long; the implementation limit is *number* characters

Error.

CL_NEWLIB, Program library *directory_specification* was created or last converted by a newer *compiler* version (*version-number*) and cannot be used by the current *compiler* version (*version-number*)

Error.

CL_NEWUNIT, *Entity* was compiled by a newer compiler (*version-number*) and cannot be used by the current compiler (*version-number*)

Error.

CL_NEWUNIT1, *Entity* was compiled by a newer compiler (*version-number*) and cannot be used by the current compiler (*version-number*)

Informational.

CL_NEWVEREXI, Parent library contains a newer version of *entity* compiled *date_time*; not merged

Error.

CL_NOENTERED, *Entity* was made visible by ENTER UNIT command, and cannot be compiled or recompiled

Error.

CL_NOFILELOADED, No file loaded

Warning.

CL_NOMAINUNIT, /NOMAIN qualifier specified, but there are no files in the command line that might include an image transfer address

Error.

CL_NOOBJFORUNIT, No object file for *entity*

Informational.

CL_NOSEARCH, No source search list (ADA$SOURCE) is defined

Informational.

CL_NOSOURCE, Source file for *entity* not found

Error.

CL_NOTADALIB, *File_specification* is not a valid Ada program library

Error.

CL_NOTADALIB_V1, *File_specification* is not a valid Version 1.n Ada program library

Error.

CL_NOTFORBOD, *Entity* cannot have a non-Ada body

Error.

CL_NOUNIREC, No units specified for RECOMPILE

Informational.

CL_NOWILDNAME, Unit names cannot have wildcard characters in this context

Error.

CL_OBSENTUNI, *Entity* has been recompiled in *file_specification* and must be reentered

Error.

CL_OBSLIB, Program library *directory_specification* was created or last converted by an obsolete *compiler* version (*obs_version*) and cannot be used by the current *compiler* version (*version-number*); use the ACS CONVERT LIBRARY command to convert the library to the current version

Error.

CL_OBSLIBUNI, Obsolete library units are detected

Error.

CL_OBSUNIBOD, Obsolete *entity* ignored

Informational.

CL_OBSUNIT, *Entity* was compiled by an obsolete compiler (*version-number*) and cannot be used by the current compiler (*version-number*); this unit should be compiled using the current compiler

Error.

CL_OBSUNIT1, *Entity* was compiled by an obsolete compiler (*version-number*) and cannot be used by the current compiler (*version-number*); this unit should be compiled using the current compiler

Informational.

CL_ONEMAIUNI, An executable program can have only one main unit

Error.

CL_OPENACU, Error opening *file-specification* for *entity* in *directory-specification*

Error.

CL_OPENFOR, Error opening foreign body *file_specification* for *entity* in *directory_specification*

Error.

CL_PREV20, Program library *directory_specification* was created by a VAX Ada version prior to Version 2.0 and cannot be used by the current *compiler* version (*version-number*); use the ACS CONVERT LIBRARY command to convert the library to the current version

Error.

CL_PROLIMEXC, Current program depends on more than implementation limit of *number* units [LRM F.9.5]

Error.

CL_QUANOTDEF, Qualifier *name* is illegal for this target

Error.

CL_RECOMPILE, The following units need to be recompiled:

Informational.

CL_RECOMPILE1, The following units will be recompiled:

Informational.

CL_REENTER, The following units need to be reentered:

Informational.

CL_RENAMEERR, Error renaming *file_specification*

Error.

CL_REORGFAIL, Reorganization of library *file_specification* failed

Error.

CL_REORGOK, *File_specification_or_name* reorganized

Informational.

CL_REQWRIACC, Operation requires write access to library *file_specification*

Error.

CL_RETURNED, Control returned to process *process_name*

Informational.

CL_SEARCHIS, Current source search list (ADA$SOURCE) is *search_list*

Informational.

CL_SECCONFAI, Multiple connections to the compilation library *file_specification* failed

Error. The VAX Ada program library manager may open the program library more than once for any operation. This message is most likely given when accessing a program library over DECnet. Usually, the cause of the problem is that the node containing the library is not accepting any more links.

CL_SECOPNFAI, Multiple opens of the compilation library *file_specification* failed

Error. The VAX Ada program library manager may open the program library more than once for any operation. This message is most likely given when accessing a program library over DECnet. Usually, the cause of the problem is that the node containing the library is not accepting any more links.

CL_SOURCEFILE, *Entity* originated in source file *file-spec*

Informational - supplemental.

CL_SRCCONACS, Source file specification *file_specification* must not contain an access control string

Fatal.

CL_SUBLIBCRE, Sublibrary *file_specification* created

Informational.

CL_SUBLIBDEL, Sublibrary *file_specification* deleted

Informational.

CL_SUBMITTED, *Job_name*

Informational.

CL_SUBNOTFOU1, Separate *entity* not found in library

Informational.

CL_SUBNOTFOU2, Separate *entity* not found in library

Error.

CL_SUBNOTLIB, *File_specification* is a sublibrary, not a library

Error.

CL_SUBUNIDUP, Subunit name *name* conflicts with prior stub *name* of unit *name* [LRM 10.2(5)]

Error.

CL_SUBUNIDUPSUP, The simple names of all subunits that have the same ancestor library unit must be distinct [LRM 10.2(5)]

Informational - supplemental.

CL_SYSLIBMIS, Program library *directory_specification* specifies a target that is not supported (SYSTEM.SYSTEM_NAME = *name*) by the current *compiler* version (*version-number*)

Error. Either use the compiler or program library manager that supports the specified target or redefine the value of the program library characteristic SYSTEM_NAME to specify a supported target. You can redefine SYSTEM.SYSTEM_NAME by compiling the pragma SYSTEM_NAME or by entering the ACS SET PRAGMA/SYSTEM_NAME command

CL_SYSNOTDEF, Logical name SYS$NODE is needed, but it is not defined

Error.

CL_TARNOTDEF, The TARGET link option is not defined

Error.

CL_TGTMISMATCH2, *Name* is compiled for an unknown target [LRM 13.7(11)]

Informational.

CL_UNARECONSNAM, Unable to reconstruct full Ada name for subunit with library name *name*

Warning.

CL_UNDEFMEMSIZ, Memory size *name* is not defined

Error.

CL_UNDEFSYSNAM, SYSTEM.NAME *name* is not defined for this target

Error.

CL_UNIALREXI, Unit *name* already exists in library; not replaced

Error.

CL_UNICIRDEP, Circular dependence among the following units:

Error.

CL_UNIDEPDFL1, *Name* depends on pragma LONG_FLOAT(D_FLOAT), but that is not the current definition for predefined type LONG_FLOAT [LRM 3.5.7a]

Informational.

CL_UNIDEPGFL1, *Name* depends on pragma LONG_FLOAT(G_FLOAT), but that is not the current definition for predefined type LONG_FLOAT [LRM 3.5.7a]

Informational.

CL_UNIDEPMEM1, *Name* depends on SYSTEM.MEMORY_SIZE, which has been redefined [LRM 13.7(11)]

Informational.

CL_UNIDEPNAM1, *Name* depends on SYSTEM.SYSTEM_NAME, which has been redefined [LRM 13.7(11)]

Informational.

CL_UNIELAORD, Compilation units in elaboration order

Informational.

CL_UNILIMEXC, Current compilation unit requires more than implementation limit of *number* units [LRM F.9.5]

Error.

CL_UNINOTENT, Unit *name* is not an entered unit

Error.

CL_UNINOTFOU, Unit *name* not found in library

Error.

CL_UNINOTREE, *Entity* not found in *file_specification*

Error.

CL_UNKREFLOP, *Entity* references *entity*, which is not in the library

Informational.

CL_UNKREFUNI, *Name* depends on *entity*, which is not in the library

Error.

CL_USESETLIB, Current program library undefined; use the ACS SET LIBRARY command

Error.

CL_VFYACU, Error in *file_specification* for *entity*

Error.

CL_VFYCORR, *File_specification_or_name* verified and repaired

Warning.

CL_VFYERR, *File_specification_or_name* has uncorrected errors

Error.

CL_VFYFIL, *File_specification* is not cataloged in library *file_specification*

Error.

CL_VFYNDX, Index entry for *name* has illegal format

Error.

CL_VFYOK, *File_specification_or_name* verified

Informational.

CL_VFYPRO, Inconsistent file protection *file_specification*

Error.

CL_VFYQUIT, Verification of *file_specification* not completed

Error.

CL_VFYREP, Use the ACS VERIFY/REPAIR command to revalidate *file_specification*

Error.

CL_VFY_RECOMPIL, Units with inaccessible files are obsolete. If repair (VERIFY/REPAIR) is not possible, then recompilation of these units is necessary; after entering a VERIFY/REPAIR command, the CHECK command will show any obsolete units

Informational.

CL_WARDURCOM, Warnings during compilation

Warning.

CL_WARDURLIN, Warnings during link

Warning.

Appendix H

Run-Time Diagnostic Messages

VAX Ada diagnostic messages generated by the Ada run-time library are presented in this appendix.

The messages are listed in alphabetical order by ident. The ident is followed by the message text, the severity level of the message, and a more detailed description of the message text. A cross-reference of the form "See also" refers you to the message for the most general category for this type of error. For example, the explanation at INTDATCOR includes the reference, "See also PROGRAM_ERROR."

H.1 Diagnostic Message Format

The general format of a run-time diagnostic message is as follows:

```
%Facility_code-Severity_code-Ident-Message_text
```

Facility_code
Is a four-letter code (ADA) that identifies the VAX Ada run-time library.

Severity_code
Is a letter (F, E, W, or I) that indicates the severity of the message. The meaning of these severity codes is discussed in Section H.2.

Ident
Is a name that uniquely identifies the message.

Message_text
Is a description of the event that has taken place. Italicized items in the message text in this appendix indicate items that are replaced with specific information when the message is actually generated.

H.2 Diagnostic Message Severity Codes

A VAX Ada run-time diagnostic message contains one of four codes—F, E, W, I—which indicate the severity of the error:

- **F** indicates a fatal error. An F-level message indicates that the intended request cannot be executed.
- **E** indicates a user error. An E-level message is often supplemented with informational (I-level) messages that give additional information about the error.
- **W** indicates a warning message. A command may have performed some, but not all, of your request, and you may have to verify the command output.
- **I** indicates an informational message. An I-level message often contains supplementary information about a preceding or otherwise related E-level error.

H.3 VAX Ada Run-Time Diagnostic Messages

ALICOLILL, Requested alignment for a collection is illegal

Fatal. The VAX Ada run-time library was asked to allocate a collection on a storage boundary that is not supported by the dynamic memory allocation routines (LIB$GET_VM).

The most likely cause of this error is an erroneous value specified on an alignment clause.

User Action. Check the value specified for the alignment clause. See also PROGRAM_ERROR.

ALREADY_OPEN, File is already open

Fatal. See also STATUS_ERROR.

AMBKEYFORM, Ambiguous keyword in FORM parameter

Informational. A keyword in the FORM parameter of a CREATE or OPEN operation has not been specified with enough characters to distinguish it from another keyword acceptable in this context. Note that VAXELN Ada accepts FORM parameter values that

are different from VMS RMS File Definition Language (FDL) statements.

User Action. Replace the keyword, specifying enough characters to make it unique.

ASTDELTER, An AST was delivered, but the task is terminated

Fatal. In VAX Ada, asynchronous system traps (ASTs) are handled by using the AST_ENTRY pragma and attribute to transform the delivery of an AST into a special kind of entry call. In this case, the task entry to which the AST was delivered belongs to a terminated task.

Note that this situation cannot be detected in all cases. In particular, it cannot be detected if the immediate master upon which the task depends has also terminated.

This error raises an exception declared by the VAX Ada run-time library. Because there is no reasonable exception handler for this case, the exception is allowed to propagate so that it can produce a traceback, or so that you can diagnose the error if you are executing the program under the control of the debugger.

User Action. Determine why a task that was to receive an AST entry call was terminated when the AST was delivered. See also PROGRAM_ERROR.

ASTNOTCAL, The task named in an AST_ENTRY attribute is not callable

Fatal. The AST_ENTRY attribute was invoked for an entry in a task that is completed and therefore cannot receive the AST.

User Action. Keep the task from becoming completed or do not use the AST_ENTRY attribute on an entry of a completed task. See also PROGRAM_ERROR.

ASTPKTQUO, The AST packet pool has been exhausted

Fatal. The pool of space from which the VAX Ada run-time library allocates AST packets for the AST_ENTRY attribute has been exhausted. The ASTs being delivered are not accepted quickly enough by the task entries that have been designated to handle them.

User Action. Make tasks receiving AST entry calls accept the entries more rapidly, perhaps by raising the priority of such tasks. You can also increase the pool of space from which AST packets are allocated by calling the VAX Ada run-time library routine

SYSTEM_RUNTIME_TUNING.EXPAND_AST_PACKET_POOL. See also PROGRAM_ERROR.

ATTUNWREN, An attempt was made to unwind a rendezvous in progress

Fatal. The condition handler that was established by the VAX Ada run-time library to monitor exceptions propagating from a rendezvous between tasks has been called with the SS$_UNWIND condition, but the rendezvous is still in progress.

The VAX Ada run-time library cannot signal this error because signaling during an unwind is forbidden by the VMS operating system. The program is forced to exit after displaying this error message.

User Action. The most likely cause of this error is an error in a call to the VMS SYS$UNWIND system service during the rendezvous. Check any non-Ada code called by the accepting task to determine if one of its handlers is requesting too deep an unwind. See also PROGRAM_ERROR.

ATTUNWTAS, Attempting to unwind the first stack frame of a task

Fatal. The first frame of a task is created by the VAX Ada run-time library and is not normally unwound (that is, it is never removed from the stack using the VMS SYS$UNWIND system service). This error condition is raised if the SYS$UNWIND system service is called to unwind this frame.

The VAX Ada run-time library cannot signal this error because signaling during an unwind is forbidden by the VMS operating system. The program is forced to exit after displaying the error message.

User Action. The most likely cause of this error is an error in a call to the VMS SYS$UNWIND system service. Check any non-Ada code called by the task to determine if one of its handlers is requesting too deep an unwind. See also PROGRAM_ERROR.

CONSTRAINT_ERRO, CONSTRAINT_ERROR

Fatal. This predefined exception is raised upon an attempt to violate a range constraint, an index constraint, or a discriminant constraint; upon an attempt to use a record component that does not exist for the current discriminant values; or upon an attempt to use a selected component, an indexed component, a slice, or an attribute of an object designated by an access value, if the object does not exist because the access value is null.

In response to Ada interpretation AI-00387, VAX Ada also raises this exception for integer overflow, floating-point overflow, and integer and floating-point division by zero. This exception is not raised by floating-point underflow (floating-point underflow is not defined as an exception in VAX Ada); underflow can be handled as an imported VAX condition.

DATA_ERROR, DATA_ERROR

Fatal. This predefined exception is raised by a TEXT_IO GET procedure if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required type or subtype. This exception may also be raised in any input operation (using any of the input-output packages) that would result in overflowing the item being written to.

DEVICE_ERROR, DEVICE_ERROR

Fatal. This predefined exception is never raised by VAX Ada. See also USE_ERROR.

DURNOTRAN, Computed duration is not in the range of the type DURATION

Fatal. See also TIME_ERROR.

END_ERROR, END_ERROR

Fatal. This predefined exception is raised by an attempt to skip (read past) the end of a file.

ERRONEOUS, Program is erroneous

Fatal. An inconsistency was detected at run time that indicates that the program is erroneous. Appended messages give more information about the error.

User Action. Follow the recommendations given by the appended messages.

EXCCOP, Exception was copied at a raise or accept statement

Fatal. This is the first in a series of exception messages that are issued when an exception (signal argument list) has been copied. Exception copying occurs at a raise statement without an exception name, or when an exception is propagating out of a rendezvous into the calling task.

VAX Ada ignores this first message when matching the exception to a choice in an exception handler. The purpose of this message is to prevent non-Ada condition handlers from mishandling the copied exception.

EXCCOPLOS, Exception was copied at a raise or accept statement, but some details were lost

Fatal. This is the first message in a series of exception messages that are issued when an exception (signal argument list) has been copied and some detailed information has been lost. Exception copying occurs at a raise statement without an exception name, or when an exception is propagating out of a rendezvous into the calling task. The lost information in the exception messages was replaced by zeros (that is, some FAO arguments were zeroed) to avoid copying a pointer into a stack that no longer exists.

VAX Ada ignores this first message when matching the exception to a choice in an exception handler. The purpose of this message is to prevent non-Ada condition handlers from mishandling the copied exception.

EXCDUREXC, An exception occurred in the VAX Ada run-time library while handling an exception

Fatal. An exception was propagated out of the first frame of a task or the main program while the task or main program was already in the process of terminating because of a prior exception.

Because there is no reasonable exception handler for this case, the exception is allowed to propagate so that it can produce a traceback, or so that you can diagnose the error if you are executing the program under the control of the debugger.

User Action. The most likely cause of this error is that the stack has overflowed and the overflow was not detected. Use the debugger to determine what caused the original exception that caused the task or main program to become terminated. Eliminating this exception is likely to also eliminate the exception during exception handling. Also, try enabling Ada checks to detect the error sooner. See also PROGRAM_ERROR.

EXCEPTION, Exception *ident*

Fatal. An exception that was declared in an exception declaration located somewhere in the Ada program was raised.

EXISTENCE_ERROR, The element does not exist

Fatal. This predefined exception is raised when the element to be read cannot be found in a relative or indexed file during the execution of a READ or READ_EXISTING procedure.

FAC_MODE_MISMAT, The file access does not allow the new mode

Informational. The file access attributes specified for the file do not match the mode desired for the file in a CREATE, OPEN, or RESET operation. See also USE_ERROR.

FAIMODTIM, Unable to modify time-slice setting

Fatal. An error occurred when the VAX Ada run-time library was calling a system service to set up time slicing. The most likely cause is that the system AST quota has been exceeded. Appended messages give more information on the error.

User Action. Observe the appended message to determine why the system service failed. See also PROGRAM_ERROR.

FAISETTIM, Unable to request another time-slice AST

Fatal. The error occurred when the VAX Ada run-time library called the VMS SYS$SETIMR system service to schedule the next time-slice AST. An appended message gives the reason for the error. See also PROGRAM_ERROR.

User Action. Examine the appended message to determine why the VMS SYS$SETIMR system service failed. If it failed because of an exceeded quota (SS$_EXQUOTA), then the most likely cause of this error is that the value of your process's AST queue limit (ASTLM) parameter was exceeded. Determine if your program has generated many ASTs while AST delivery has been disabled by a call to the VMS SYS$SETAST system service. If there are no such program errors, then ask your system manager to increase the value of your ASTLM parameter (a UAF parameter). Then try your program again. See the description of SYS$SETIMR in the *VMS System Services Reference Manual* for additional situations that can cause a status of SS$_EXQUOTA to be returned.

FATINTERR, Fatal internal error in the VAX Ada run-time library

Fatal.

User Action. Submit a Software Performance Report (SPR) to Digital, including a machine-readable copy of your program, data, and a sample execution showing the problem.

INSSPAALL, Insufficient space to allocate from a collection

Fatal. An explicit (or implicit) allocator cannot allocate from a collection. See also STORAGE_ERROR.

INSSPACOL, Insufficient space to create a collection

Fatal. See also STORAGE_ERROR.

INSSPATAS, Insufficient space to create a task

Fatal. See also STORAGE_ERROR.

INTDATCOR, Internal data in the VAX Ada run-time library is corrupted

Fatal. The data corruption may have been caused by a VAX Ada error or by your program.

User Action. If you cannot determine the source of the error, please submit a Software Performance Report (SPR) to Digital, including a machine-readable copy of your program, data, and a sample execution showing the problem. See also PROGRAM_ERROR.

INVVALFORM, Invalid attribute value in FORM parameter

Informational. The FORM parameter of a CREATE or OPEN operation contains an attribute value that is not legal for the attribute's keyword. Note that VAXELN Ada accepts FORM parameter values that are different from VMS RMS File Definition Language (FDL) statements.

User Action. Either the keyword or its attribute's value is incorrect. Replace the invalid attribute value with a legal value, or replace the attribute's keyword with one for which the attribute's value is legal.

KEYSIZERR, Size of the key is not a multiple of 8 bits

Fatal. A read operation from an indexed file has specified a key that is not a multiple of 8 bits. See also KEY_ERROR.

KEY_ERROR, Key is inappropriate for the file

Fatal. This predefined exception is raised in an indexed file if the key has been changed or duplicated and changes or duplicates are not permitted. This exception is also raised if a read operation from an indexed file has specified a key that is not a multiple of 8 bits.

KEY_MISMATCH, The file key does not match the key value specified in the FORM parameter

Informational. The OPEN operation has detected that the key specification asserted in the FORM string does not match the key specification of the file being opened. See also USE_ERROR.

LAYOUT_ERROR, LAYOUT_ERROR

Fatal. This predefined exception is raised by the TEXT_IO COL, LINE, or PAGE operations if the value returned exceeds COUNT'LAST; on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, respectively (excluding the unbounded cases); by an attempt to write too many characters to a string with a PUT procedure; and in item operations of the mixed input-output packages when a GET_ITEM or PUT_ITEM operation results in reading or writing beyond the file buffer.

LINEXCMRS, Line will exceed external file's maximum record size

Informational. The TEXT_IO operation will overflow the maximum record size of the external file. See also USE_ERROR.

LOCK_ERROR, The element is locked

Fatal. This predefined exception is raised by a READ or READ_EXISTING procedure if the result is a locked record error in a relative or indexed file.

MAXLINEXC, Maximum line length exceeded

Informational. The line length specified by the TEXT_IO.SET_LINE_LENGTH procedure exceeds the maximum record size of the file. See also USE_ERROR.

MISKEYFORM, Missing or unrecognized keyword in FORM parameter

Informational. The FORM parameter of a CREATE or OPEN procedure contains an illegal keyword value. Note that VAXELN Ada accepts FORM parameter values that are different from VMS RMS File Definition Language (FDL) statements.

User Action. Supply the missing keyword or correct the illegal keyword.

MODE_ERROR, MODE_ERROR

Fatal. This predefined exception is raised by an attempt to read from, or test for the end of, a file whose current mode is OUT_FILE, and also by an attempt to write to a file whose current mode is IN_FILE. In the case of TEXT_IO operations, the exception MODE_ERROR is also raised by specifying a file whose current mode is OUT_FILE in a call of SET_INPUT, SKIP_LINE, END_OF_LINE, SKIP_PAGE, or END_OF_PAGE; and by specifying a file whose current mode is IN_FILE in a call of SET_OUTPUT, SET_LINE_LENGTH, SET_PAGE_LENGTH, LINE_LENGTH, PAGE_LENGTH, NEW_LINE, or NEW_PAGE.

MRN_MISMATCH, The file maximum record number does not match the maximum record number specified in the FORM parameter

Informational. The OPEN operation has detected that the maximum record number asserted in the FORM parameter does not match the maximum record number of the file being opened. See also USE_ERROR.

MRS_MISMATCH, The file maximum record size does not match the maximum record size specified in the FORM parameter

Informational. The OPEN operation has detected that the maximum record size asserted in the FORM parameter does not match the maximum record size of the file being opened. See also USE_ERROR.

NAME_ERROR, NAME_ERROR

Fatal. This predefined exception is raised by a call of a CREATE or OPEN procedure if the string given for the parameter NAME does not identify an external file. For example, this exception is raised if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string.

NON_ADA_ERROR, NON_ADA_ERROR

Fatal. This exception is declared in the package SYSTEM. When used as a choice in an Ada exception part, NON_ADA_ERROR matches itself or any VMS (that is, non-Ada) exception. It allows the treatment of non-Ada conditions as a special subclass of Ada exceptions.

NOTASTLEV, *Name* cannot be called at AST level

Fatal.

User Action. Modify your program so that the specified operation is no longer called from an AST service routine. See also PROGRAM_ERROR.

NOT_OPEN, File is not open

Fatal. See also STATUS_ERROR.

NUMERIC_ERROR, NUMERIC_ERROR

Fatal. In response to Ada interpretation AI-00387, VAX Ada raises NUMERIC_ERROR only when it is explicitly raised with a raise statement. Wherever the Ada language standard requires that NUMERIC_ERROR be raised, CONSTRAINT_ERROR is raised instead.

ORG_MISMATCH, The file organization does not match the organization specified in the FORM parameter

Informational. The OPEN operation has detected that the VMS RMS organization asserted in the FORM parameter does not match the organization of the file being opened. See also USE_ERROR.

PACNUMILL, Illegal number of AST packets was requested

Fatal. The number of AST packets requested by the SYSTEM_RUNTIME_TUNING.EXPAND_AST_PACKET_POOL procedure is either less than zero or, when added to the number of existing AST packets, exceeds the number of AST packets allowed by the VAX Ada run-time library.

User Action. Modify your program to pass a correct value to the SYSTEM_RUNTIME_TUNING.EXPAND_AST_PACKET_POOL procedure. If you need more than the current limit of AST packets then make tasks receiving AST entry calls accept them more rapidly, perhaps by raising the priority of such tasks. See also PROGRAM_ERROR.

PROGRAM_ERROR, PROGRAM_ERROR

Fatal. This predefined exception is raised upon an attempt to call a subprogram, to activate a task, or to elaborate a generic instantiation, if the body of the corresponding unit has not yet been elaborated. This exception is also raised if the end of a function is reached; or during the execution of a selective wait that has no else part, if this execution determines that all alternatives are closed. Finally, this exception may be raised upon an attempt to execute an action that is erroneous.

Additional messages are sometimes appended to this exception to further identify the reason why it was raised.

RAT_MISMATCH, The file record attribute does not match the record attribute specified in the FORM parameter

Informational. The OPEN operation has detected that the record attribute asserted in the FORM parameter does not match the record attribute of the file being opened. See also USE_ERROR.

RECNOTPOS, Program is erroneous, error recovery by exception handling is not possible

Fatal. An error that cannot be corrected by an Ada exception handler has been detected at run time. Either there is no appropriate handler or the error condition would remain after the exception was handled. The program is presumed to be erroneous.

Typically, the cause of such an error is that the program has become corrupted because it suppresses Ada checking, it misuses the AST_ENTRY attribute, or because it improperly uses imported non-Ada subprograms (such as system services).

Appended messages give more information about the error.

User Action. Determine from the appended messages what the program did to cause the VAX Ada run-time library to fail. Also, try enabling checking in the Ada program, and carefully investigate the use of imported subprograms and the AST_ENTRY attribute. See also PROGRAM_ERROR.

RFM_MISMATCH, The file record format does not match the record format specified in the FORM parameter

Informational. The OPEN operation has detected that the record format asserted in the FORM parameter does not match the record format of the file being opened. See also USE_ERROR.

SELALTCLS, All select alternatives are closed and there is no else part

Fatal. See also PROGRAM_ERROR.

SIGVECIMP, Signal vector is improperly formatted—one or more FAO arguments are missing

Fatal. While copying an exception, the VAX Ada run-time library has detected that the signal arguments are improperly formatted. Most likely an FAO argument count is incorrect.

If you cannot determine the source of the error, submit a Software Performance Report (SPR) to Digital, including a machine-readable copy of your program, data, and a sample execution showing the problem.

STAOVF, Attempted stack overflow was detected

Fatal. See also STORAGE_ERROR.

STATUS_ERROR, STATUS_ERROR

Fatal. This predefined exception is raised by an attempt to operate upon a file that is not open, and by an attempt to open a file that is already open.

STORAGE_ERROR, STORAGE_ERROR

Fatal. This predefined exception is raised in any of the following situations: when the dynamic storage allocated to a task is exceeded; during the evaluation of an allocator, if the space available for the collection of allocated objects is exhausted; or during the elaboration of a declarative item, or the execution of a subprogram call, if storage is not sufficient.

Appended messages give more information about the error.

User Action. Typically, two situations raise this exception: the program has no more free virtual pages for any allocations, or an attempt was made to exceed the amount of storage specified in a length clause (in other words, the value specified for T'STORAGE_SIZE was exceeded).

In the first situation, see if the program has an error that causes a large number of allocators to be evaluated; for example, an infinite loop containing allocator evaluations. If the program has no error, ask your system manager to consider increasing the value of the SYSGEN VIRTUALPAGECNT parameter (maximum number of virtual pages parameter) on your system.

In the second situation, consider changing the value of a task or access type length clause STORAGE_SIZE attribute designator.

Use the appended message to further determine the reason for the exception.

STOSIZILL, Requested value of STORAGE_SIZE for a collection is illegal

Fatal. Typically, this error can occur if the program specifies an illegal value for a length clause STORAGE_SIZE attribute designator, and compiler checks have been suppressed so that the illegal value is not detected at compile time.

User Action. Check the STORAGE_SIZE value for the appropriate access type. Try recompiling the program (or compilation unit) with checking enabled. See also PROGRAM_ERROR.

SUBNOTELA, The body of the called subprogram has not yet been elaborated

Fatal. See also PROGRAM_ERROR.

SYNERRFORM, Syntax error in FORM parameter

Informational. The FORM parameter of a CREATE or OPEN procedure cannot be parsed because it contains a syntax error. Note that VAXELN Ada accepts FORM parameter values that are different from VMS RMS File Definition Language (FDL) statements.

User Action. Correct the syntax error in the FORM parameter.

TASCOMACT, A task completed during its activation

Fatal. See also TASKING_ERROR.

TASKING_ERROR, TASKING_ERROR

Fatal. This predefined exception is raised when exceptions arise during intertask communication.

Appended messages give more information about the error.

TASNOTCAL, The task named on an entry call is not callable

Fatal. See also TASKING_ERROR.

TASNOTELA, A task's body was not elaborated before its activation

Fatal. See also TASKING_ERROR.

TASSTOSMA, Requested STORAGE_SIZE for a task is illegal

Fatal.

User Action. Typically, this error can occur if the program specifies an illegal value for a length clause STORAGE_SIZE attribute designator, and compiler checks have been suppressed so that the illegal value is not detected at compile time.

Check the STORAGE_SIZE value for the appropriate task type. Try recompiling the program (or compilation unit) with checking enabled. See also PROGRAM_ERROR.

TASTERAST, A task is terminating with an AST pending

Fatal. A task that should have waited for an AST to be delivered is terminating. This situation is erroneous because the task's stack must not be deallocated (as it would be at task termination) while a system service is possibly accessing the stack.

User Action. Determine why the task that was to wait for an AST is terminating. Use the debugger to determine if the task is being terminated because of an exception. See also PROGRAM_ERROR.

TIMERFAIL, Insufficient quota for call to SYS$SETIMR at delay statement

Fatal. A status of SS$_EXQUOTA was returned by the VMS SYS$SETIMR system service when it was called by the VAX Ada run-time library as part of its implementation of a delay statement.

User Action. The most likely cause of this error is that the value of your process's AST queue limit (ASTLM) parameter was exceeded. Determine if your program has generated many ASTs while AST delivery has been disabled by a call to the VMS SYS$SETAST system service. If there are no such program errors, then ask your system manager to increase the value of your ASTLM parameter (a UAF parameter). Then try your program again. See the description of SYS$SETIMR in the *VMS System Services Reference Manual* for additional situations that can cause a status of SS$_EXQUOTA to be returned. See also PROGRAM_ERROR.

TIME_ERROR, TIME_ERROR

Fatal. This predefined exception can be raised by the TIME_OF, "+", and "-" operations in the predefined package CALENDAR.

TIMPARNOT, TIME_OF parameters do not form a proper date

Fatal. See also TIME_ERROR.

TOOMANENT, Task type has too many entries

Fatal. The total number of entries (including members in entry families) for some task type exceeds the value of the constant MAX_INT declared in the package SYSTEM.

User Action. Reduce the total number of entries, including entry family members. Perhaps move some of the entries to a different task type. See also PROGRAM_ERROR.

UNSUPPORTED, The input-output package does not support the intended operation

Informational. For example, some input-output packages support only certain RMS file organizations. See also USE_ERROR.

USE_ERROR, USE_ERROR

Fatal. This predefined exception is raised when an attempted operation is not possible for reasons that depend on characteristics of the external file. For example, this exception can be raised by a CREATE procedure, if the given mode is OUT_FILE, but the form parameter specifies an input only device.

YEANOTRAN, Computed year is not in the range of subtype YEAR_NUMBER

Fatal. The subtype YEAR_NUMBER is declared in the package CALENDAR. See also CONSTRAINT_ERROR, PROGRAM_ERROR, and TIME_ERROR.

ZONECORR, The "zone" used to implement the collection for the object being allocated or deallocated has been corrupted

Fatal. The VAX Ada run-time library implements collections using the VMS Run-Time Library LIB$ memory allocation operations. In particular, Ada collections are implemented as zones. This error code is returned when LIB$GET_VM or LIB$FREE_VM fails because the zone from which the object is being allocated or deallocated has been corrupted.

User Action. Make sure that your program is not corrupting the zone. For example, be sure that your program is not calling an instantiation of the generic procedure UNCHECKED_DEALLOCATION to deallocate an object that has already been deallocated. One way this can happen is when two access variables designate the same object, and an instantiation of UNCHECKED_DEALLOCATION is called twice, once for each access variable.

Also, if your program is written in more than one language, make sure your program is not allocating an object in one language and deallocating it in another. In addition, ensure that your program has not disabled array indexing checks; writing at random memory addresses can also cause the heap to become corrupted. See also PROGRAM_ERROR.

Appendix I

Reporting Problems

If an error occurs while you are using VAX Ada and you believe that the error is caused by a problem with VAX Ada, take one of the following actions:

- If you purchased VAX Ada within the past 90 days and you think the problem is caused by a software error, you can submit a Software Performance Report (SPR).
- If you have a Basic or DECsupport Software Agreement, you should call your Customer Support Center. With these services, you receive telephone support that provides high-level advisory and remedial assistance. For more information, contact your local Digital representative.
- If you have a Self-Maintenance Software Agreement, you can submit a Software Performance Report (SPR).

If you find an error in the VAX Ada documentation, you should fill out and submit the Reader's Comments form at the back of the document in which the error was found. Specify the section and page number where the error was found.

When you prepare to submit an SPR, please do the following:

1. Describe as accurately as possible the state of the system and the circumstance when the problem occurred. Include in the description the version number of VAX Ada being used. Demonstrate the problem with specific examples.
2. Reduce the problem to as small a size as possible.
3. Remember to include listings of any command files, relevant data files, and so on.
4. Provide a listing of the program.

5. If the program is longer than 50 lines, submit a copy of the program on machine-readable media (floppy diskette or magnetic tape). If necessary, also submit a copy of the program library used to build the application. Use the VMS Backup Utility to copy the program library to the machine-readable media. All media will be returned to you when the SPR is answered.
6. Report only one problem per SPR. This will facilitate a more rapid response.
7. Mail the SPR package to Digital.

Experience shows that many SPRs do not contain sufficient information to duplicate or identify the problem. Complete and concise information will help Digital give accurate and timely service to software problems.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local DIGITAL subsidiary or approved distributor
Internal[1]	————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Developing Ada Programs on VMS Systems
AA–EF86B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

t Along Dot' 'Line

Reader's Comments

Developing Ada Programs on VMS Systems
AA–EF86B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

ut Along D Line

digital