DEC Ada

Developing Ada Programs on OpenVMS Systems

Order Number: AA–PWGYA–TK

January 1993

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

Revision/Update Information:	This revised manual supersedes *Developing Ada Programs on VMS Systems* (Order No. AA–EF86B–TE).
Operating System and Version:	VMS Version 5.4 or higher OpenVMS AXP Version 1.0 or higher
Software Version:	DEC Ada Version 3.0

Digital Equipment Corporation
Maynard, Massachusetts

February 1985
Revised, May 1989
Revised, January 1993

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

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

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

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1985, 1989, 1993.

All Rights Reserved.

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: AXP, DEC, DEC Ada, DECnet, DECset VMS, Digital, OpenVMS, ULTRIX, VAX, VAX Ada, VAX Pascal, VAXcluster, VAXELN, VAXset, VAXStation, VMS, VAX–11/780, XD Ada, and the DIGITAL logo.

ZK5577

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

2 Working with DEC Ada Program Libraries and Sublibraries

3 Working with DEC Ada Library Search Paths

4 Compiling and Recompiling DEC Ada Programs

5 Using the Professional Development Option

6 Linking Programs

7 Managing Program Development

8 Debugging DEC Ada Tasks

A ACS Command Dictionary

Examples

Figures

Tables

Preface

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

All references to VMS systems refer to OpenVMS AXP systems and OpenVMS VAX systems unless otherwise specified.

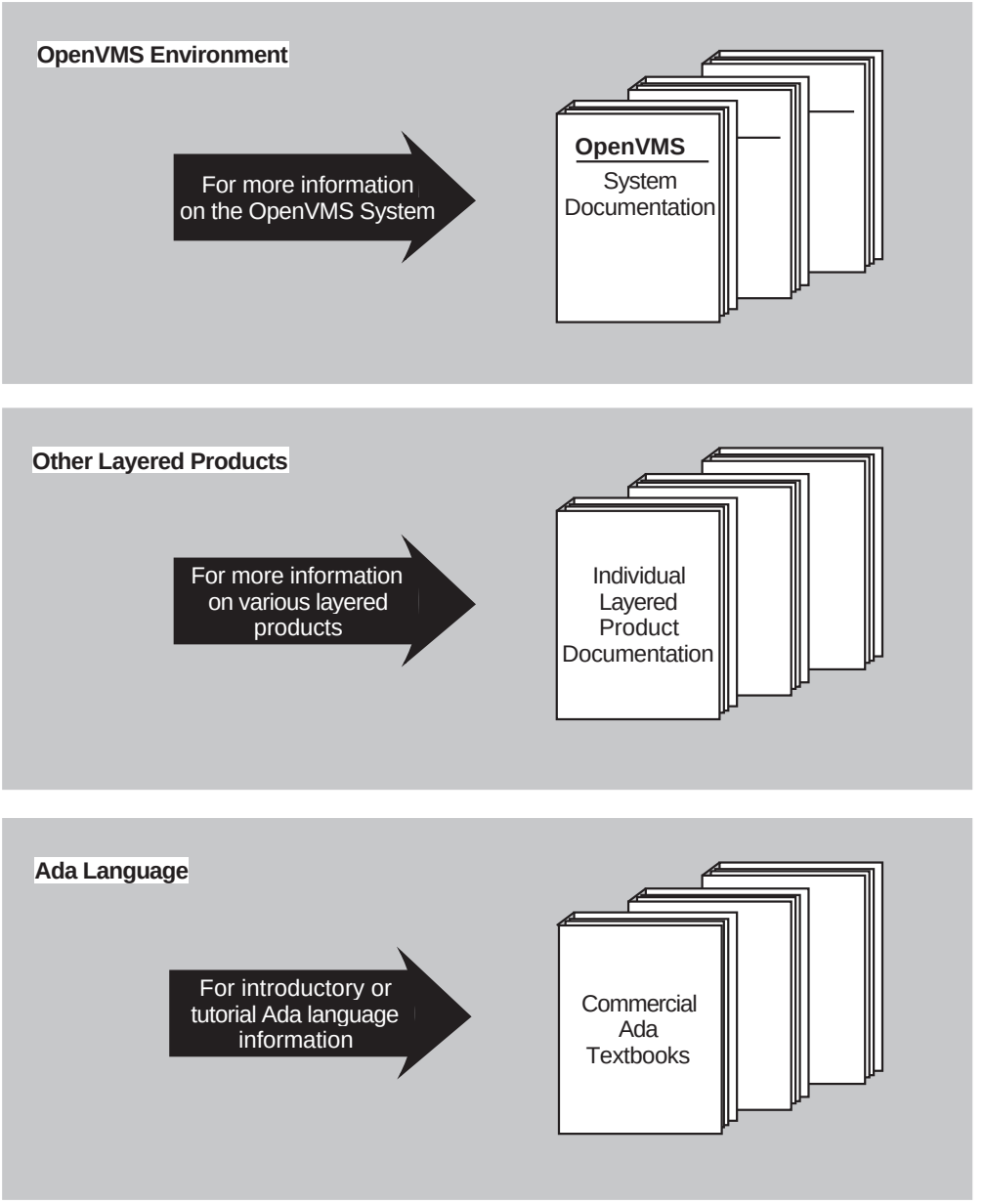
Intended Audience

This manual is intended for any programmer who needs information on compiling, linking, and executing DEC Ada programs. The reader should have a working knowledge of Ada, the Digital Command Language (DCL), and DCL command procedures.

Documentation Reading Path

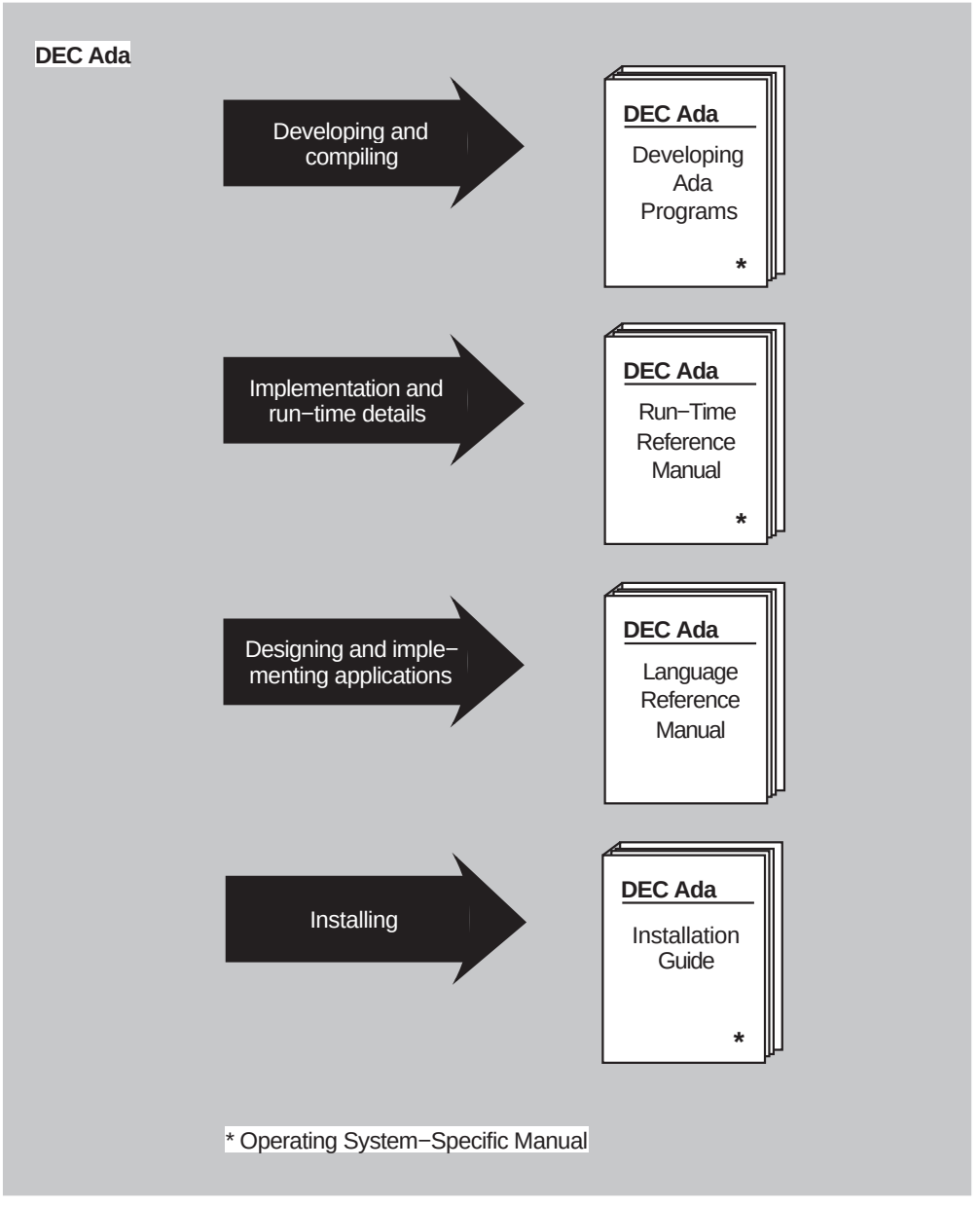
Figures 1 and 2 show the relationship of the Ada documentation set to other documentation that may be helpful.

Figure 1 Documentation Reading Path for Related Documents



ZK–5349A–2–GE

Figure 2 Documentation Reading Path for DEC Ada Documentation



ZK−5349A−1−GE

Document Structure

This manual contains the following chapters and appendixes:

- Chapter 1 provides introductory material on DEC Ada and the DEC Ada programming environment.
- Chapter 2 discusses working with DEC Ada program libraries.
- Chapter 3 discusses working with DEC Ada library search paths.
- Chapter 4 describes how to compile and recompile DEC Ada programs.
- Chapter 5 describes the features and capabilities of the Professional Development option.
- Chapter 6 describes how to link DEC Ada programs.
- Chapter 7 discusses how to manage program development.
- Chapter 8 discusses how to debug Ada tasks using the debugger.
- Appendix A describes the DEC Ada commands.
- Appendix B compares the commands for DEC Ada for ULTRIX and VMS systems.
- Appendix C provides supplemental debugging information for debugging DEC Ada programs.
- Appendix D describes how to use program design support.
- Appendix E discusses DEC Ada diagnostic messages.
- Appendix F describes how to report problems.

Conventions

Table 1 shows the conventions used in this manual.

Table 1 Conventions Used in This Manual

Convention	Description
VMS systems	Refers to OpenVMS AXP and OpenVMS VAX systems unless otherwise specified.
$	A dollar sign ($) represents the VMS DCL system prompt.

(continued on next page)

Table 1 (Cont.) Conventions Used in This Manual

Convention	Description
Return	In interactive examples, a label enclosed in a box indicates that you press a key on the terminal, for example, Return.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
`boldface monospace text`	In interactive examples, boldface monospace text represents user input.
file-spec, . . .	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates that additional parameters, options, or values can be entered.
n	A lowercase italic *n* indicates the generic use of a number.
. . .	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.
( )	In format descriptions, if you choose more than one option, parentheses indicate that you must enclose the choices in parentheses.
[expression]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{, 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.
boldface text	Boldface text indicates Ada reserved words.
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*.)

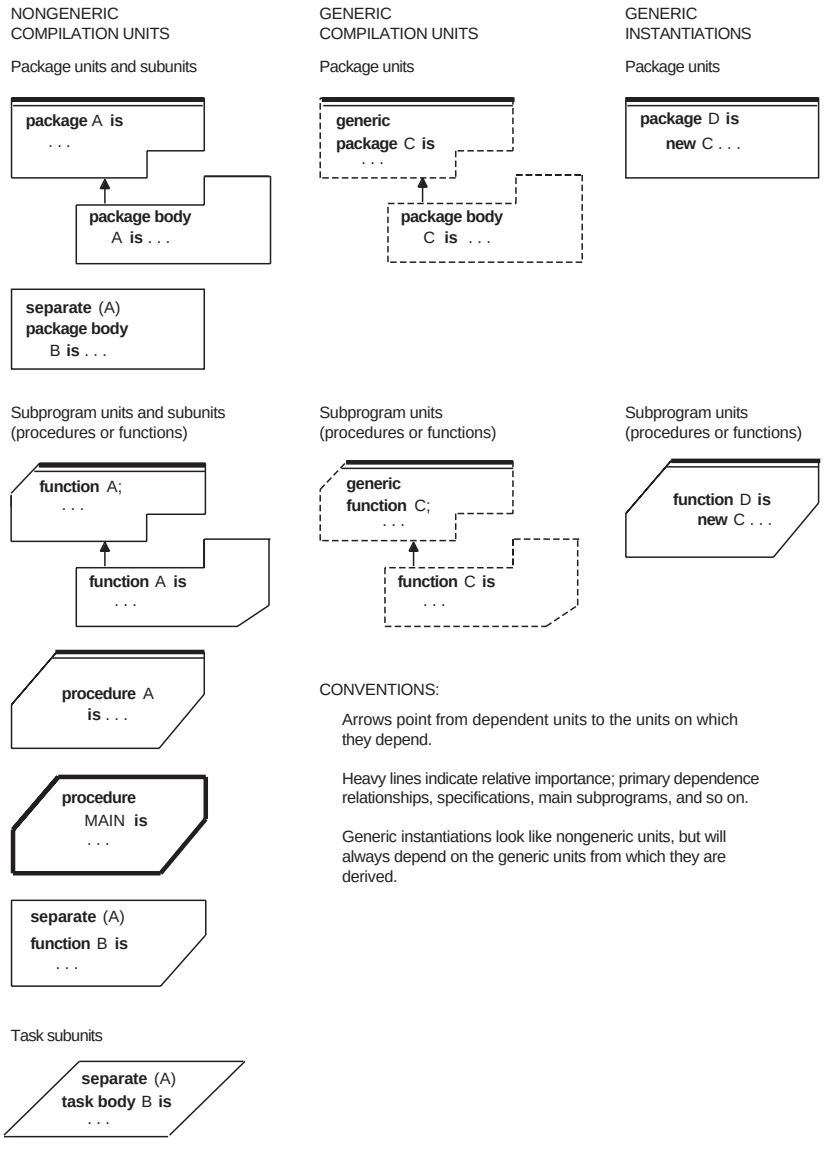
(continued on next page)

Table 1 (Cont.) Conventions Used in This Manual

Convention	Description
*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 catgegory.
UPPERCASE TEXT	Uppercase indicates the name of a command, routine, parameter, procedure, utility, file, file protection code, or the abbreviation for a system privilege.

Figure 3 explains the shapes and conventions used in figures that diagram Ada programs.

Figure 3 Figure Conventions



New and Changed Features

This release brings VAX Ada into the family of DEC Ada compilers by renaming it and improving its functionality. This release also adds a new member of the DEC Ada family by supporting Ada on the OpenVMS AXP platform.

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 2.0:

- Support for smart recompilation has been added. (See Chapter 5 for more information.) Several qualifiers have been added or changed to support smart recompilation. Specifically, the following changes have been made:
  - The /[NO]SMART_RECOMPILATION qualifier has been added to the DCL ADA and ACS CHECK, COMPILE, LOAD, RECOMPILE, and SHOW PROGRAM commands. This qualifier controls whether detailed information about unit dependences is stored in the program library and used to minimize the number of units that are considered obsolete and actually recompiled. By default (/SMART_RECOMPILATION), detailed information is stored and used to minimize recompilations.
  - The /LOG qualifier for the ACS CHECK, COMPILE, RECOMPILE and SHOW PROGRAM commands has been enhanced to report on units that may not need to be recompiled due to smart recompilation support.
- The defaults for several existing qualifiers were changed as follows:
  - The default behavior of the ACS COMPILE, LOAD, and RECOMPILE commands has changed. The new behavior is to invoke the compiler interactively (/WAIT). Prior to Version 3.0, the default behavior for these commands was to submit a batch job (/SUBMIT) to perform the compilations.

  - The default for the /[NO]PRELOAD qualifier for the ACS COMPILE command has changed. The default is now /PRELOAD.
  - The /[NO]STATISTICS qualifier has been added to the ACS CHECK, COMPILE and RECOMPILE commands. This qualifier controls whether statistical information, such as the number of obsolete units and the amount of time the recompilation requires, is displayed.

- To revert to the old behavior (prior to Version 3.0) for the DCL ADA and ACS COMPILE, LOAD and RECOMPILE commands, define the following global symbols as shown:

```
ADA == ADA/NOSMART
ACS$COM*PILE == "COMPILE/NOSMART_RECOMP/NOPRELOAD/NOSTATISTICS/SUBMIT"
ACS$REC*OMPILE == "RECOMPILE/NOSMART_RECOMPILATION/NOSTATISTICS/SUBMIT"
ACS$LO*AD == "LOAD/SUBMIT/NOSMART_RECOMPILATION"
```

- Support for library search paths has been added. (See Chapter 3.) In particular, the following changes have been added:
  - The /PATH qualifier has been added to the DCL ADA and ACS SET LIBRARY commands. This qualifier, which is also available to the ACS MODIFY LIBRARY command, allows you to specify a library search path as a parameter or qualifier value.
  - The DCL ADA command now accepts a library search path as the value of the /LIBRARY qualifier when the /PATH qualifier is also specified.
  - The ACS MODIFY LIBRARY command has been added. This command is useful for changing the default library search path of a program library.
  - The output of the ACS SHOW LIBRARY/FULL command has been enhanced with information on library search paths.
  - The /[NO]VERIFY qualifier has been added to the ACS SET LIBRARY command and controls whether the current path is evaluated and verified when the ACS SET LIBRARY command is entered. This qualifier is also available with the new ACS MODIFY LIBRARY command.

- Two qualifiers have been added that allow you to use DEC Ada as a program design processor:
  - The /[NO]DESIGN qualifier has been added to the DCL ADA, ACS COMPILE, LOAD, and RECOMPILE commands. This qualifier allows you to process Ada source files as a detailed program design. (The /DESIGN qualifier accepts the following qualifier options: [NO]COMMENTS and [NO]PLACEHOLDERS.)

- The /PROCESSING_LEVEL qualifier has been added to the ACS CHECK and SHOW PROGRAM commands. This qualifier determines the kind of obsolete units identified. (The /PROCESSING_LEVEL qualifier accepts the following qualifier options: SYNTAX, DESIGN, and FULL).

See Appendix D for more information.

- The /[NO]OBSOLETE qualifier has been added to the ACS CHECK, COMPILE, RECOMPILE, and SHOW PROGRAM commands. For the ACS CHECK and SHOW PROGRAM commands, this qualifier allows you to ask what the effect on a program or set of units would be if some specific units were made obsolete. For the ACS COMPILE and RECOMPILE commands, this qualifier allows you to force the recompilation of specific units. (The /OBSOLETE qualifier accepts the following qualifier options: UNIT, SPECIFICATION, and BODY.)

  See Section 4.5 for more information on using the /OBSOLETE qualifier.

  Note that the /[NO]OBSOLETE qualifier replaces the /[NO]DATE_CHECK and /FORCE_BODY qualifiers in this release.

- The logical name ADA$PREDEFINED is now defined using rooted directory syntax. This allows references to ADA$PREDEFINED to be independent of a specific device or directory.

1

Introduction to the DEC Ada Program Development Environment

DEC 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, DEC Ada also implements features designed to make programming in the VMS environment convenient and efficient.

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

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

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

DEC 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. DEC 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 Programming Guide* for more information on VAXELN Ada. See the XD Ada documentation for more information on XD Ada.

This chapter provides information on developing Ada programs for both the experienced programmer and the novice user. For the experienced programmer, a short section on getting started is provided. For the novice user, a step-by-step tutorial on developing Ada programs is provided. This chapter also provides an overview of the program library manager and its command language, and explains the DEC Ada conventions and terminology related to compiling, linking, and managing program libraries.

1.1 Getting Started with DEC Ada for the Experienced Programmer

If you are an experienced programmer—that is, you are familiar with Ada or programming on VMS systems—you may not need very much instruction to get started. This section provides a quick overview of the commands you will need to get an Ada program compiled and linked.

You compile and link DEC Ada programs in the context of an Ada program library, which is managed by the DEC Ada program library manager (ACS). To start using DEC Ada, enter the following ACS commands to create, initialize, and define your current program library. This series of commands assumes that your program library is a subdirectory of your current working directory.

```
$ ACS CREATE LIBRARY [.ADALIB]
$ ACS SET LIBRARY [.ADALIB]
```

Then, compile, link, and execute your program using the following commands (assuming that the name of your program is MYPROG, and that it depends on the package MYPACK):

```
$ ADA MYPACK_.ADA, MYPACK.ADA, MYPROG.ADA
$ ACS LINK MYPROG
$ RUN MYPROG
```

You can also use the following equivalent set of commands:

```
$ ACS LOAD MY*
$ ACS COMPILE MYPROG
$ ACS LINK MYPROG
$ RUN MYPROG
```

Note that when you successfully compile an Ada program, the program library manager stores the object files associated with the program in your current program library, not in your current default directory.

Once you compile an Ada compilation unit into a program library, the program library manager recognizes the unit by its unit name, not by its source file name.

1.2 Getting Started with DEC Ada for the Novice User

If you are a novice user—that is, you are unfamiliar with Ada or programming on VMS systems—you may need an overview of how to develop DEC Ada programs on VMS systems. The following sections provide detailed instructions on how to compile, link, and execute DEC Ada programs.

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

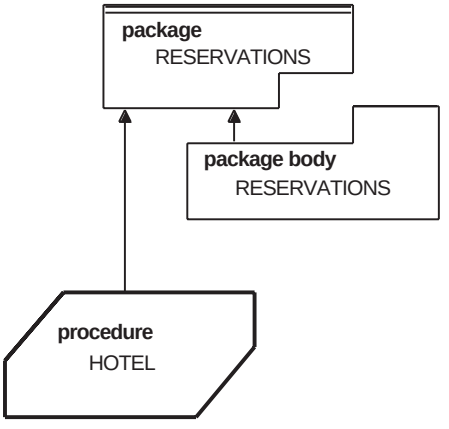
1. Create a working directory for your Ada source files, and define a current default directory for operations such as editing, debugging, and so on.
2. Create Ada source files for all of the compilation units in your program.
3. Create a program library.
4. Define a current program library for operations 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 and make changes to the source file, 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

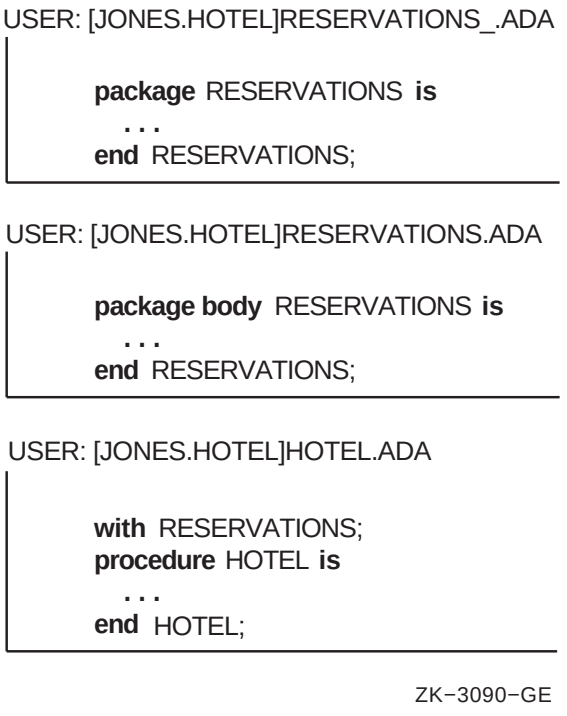


ZK–6743–GE

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 DEC Ada program library manager and the VMS Debugger.)
- The working directory and current default directory are the directory [JONES.HOTEL].
- The program library is the 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



1.2.1 Creating a Working Directory and Defining a Current Default Directory

The first steps in developing a DEC 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 previous example, these directories are usually the same.

1.2.2 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 EDT which is an interactive text editor available with the VMS operating system. Another editor is the Extensible VAX Editor (EVE), which is an interface to the VAX Text Processing Utility (DECTPU).

You can also use the DEC 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. DECTPU is part of and accessible from LSE. LSE is integrated with the DEC Source Code Analyzer (SCA) and DEC/Code Management System (CMS).

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)
- *OpenVMS EDT Reference Manual*—provides comprehensive reference information on the EDT editor
- *Guide to the DEC Text Processing Utility*—provides comprehensive reference information on DECTPU and EVE
- *Guide to Language-Sensitive Editor for VMS Systems*—provides tutorial and reference information on LSE

1.2.3 Creating a Program Library

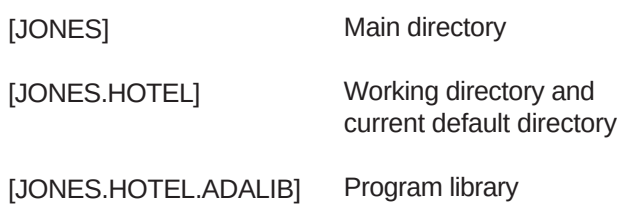
To compile or link an Ada program, you must have a *program library*. A program library is a special VMS directory that you create with the ACS CREATE LIBRARY command, specifying the name of the directory as a parameter. For example:

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

The program library holds the products of DEC 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



DEC Ada also allows you to create one or more *program sublibraries*. See Chapters 2 and 7 for more information on using sublibraries.

1.2.4 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]
```

1.2.5 Compiling the Program

To compile DEC 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 1.2.1)
- Created and set a current program library for the products of compilation (see Section 1.2.3)

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 1.4.2 and the *DEC Ada Language Reference Manual* for more information on Ada order-of-compilation rules.

1.2.6 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                16-DEC-1992 17:54:24.06    <main>

RESERVATIONS
    package specification         16-DEC-1992 17:54:09.46
    package body                  16-DEC-1992 17:54:30.80

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.

1.2.7 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 DEC Ada program library manager, which serves as the interface to the 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 1.4.1.2 and 1.4.3), 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].

1.2.8 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 the previous example.

1.2.9 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 Chapter 8 for information on debugging tasks; see the *OpenVMS Debugger Manual* for information on the debugger).

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

        OpenVMS VAX DEBUG Version 5.5

%I, language is ADA, module set to HOTEL
%I, 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 qualifier, and want to execute it without debugger control, you can enter the DCL RUN/NODEBUG command, as follows:

```
$ RUN/NODEBUG HOTEL
```

1.2.10 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 1.2.5. Alternatively, you can use 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 1.4.1.2 for more information on obsolete units, incomplete units, and recompilation.)

Because the ACS COMPILE command assumes the /PRELOAD qualifier by default, you can compile a modified set of units whose compilation order has changed in the correct order. However, if you have created new source files to add new units to your program, you must add the new 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 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 /OBSOLETE qualifier, you can use them to force the 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 4 for more information on using the ACS COMPILE and RECOMPILE commands.

1.3 Using the DEC Ada Program Library Manager

The DEC Ada program library manager is an interactive utility with ACS commands. You enter these commands 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 DEC Ada compiler and 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

1.3.1 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 7 for more information on program library management.)
- Table 1–2 summarizes compilation and linking commands. (See Chapter 4 for more information on compilation; see Chapter 6 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.
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 DEC Ada program library.
CREATE SUBLIBRARY	Creates a DEC 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.
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. The MERGE command replaces the older, obsolete versions in the parent library.

[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 1.4.1.2 and 1.4.3, respectively.

(continued on next page)

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

Command	Function
MODIFY LIBRARY	Modifies the default path of a DEC Ada program library or sublibrary.
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 FLOAT_REPRESENTATION, 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 DEC 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 DEC Ada compiler to compile the specified Ada source files.
RUN	Executes the specified executable image file.

(continued on next page)

Table 1–2 (Cont.) Compilation, Linking, and Execution Commands

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 become obsolete. Completes incomplete generic instantiations.
LOAD	Loads (partially compiles) the units in the specified 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 1.4.1.2 and 1.4.3, respectively.

Table 1–3 Additional ACS Commands

Command	Function
ATTACH	Switches control of your terminal from the current process running the DEC 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 HELP facility to obtain information about ACS commands.
SPAWN	Creates a subprocess of the current process (same as the DCL SPAWN command).

1.3.2 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 *OpenVMS User's Manual*.). 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.

1.3.3 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.

1.3.4 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 *OpenVMS User's 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.

1.3.5 Using DCL Commands with Program Libraries

Program libraries are implemented in DEC 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 7). Similarly, you may need to use the DCL BACKUP command to copy or back up a program library (see Chapter 7).

1.4 Concepts and Terminology

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

1.4.1 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.

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.

1.4.1.1 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 1.4.2), 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 7 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.

1.4.1.2 Current and Obsolete Units

Whenever a unit is compiled, any dependent unit, as defined in Section 1.4.1.1, becomes *obsolete* and must eventually be recompiled 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 1.4.2), DEC 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.

If smart recompilation is in effect, dependent units are recompiled only if they are actually affected by a change. See Chapter 5 for more information.

1.4.1.3 Unit and File-Name Conventions

While developing programs in the DEC 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 becomes 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 DEC 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
subunits		
INPUT	SCREEN_IO.INPUT	SCREEN_IO__INPUT
BUFFER	SCREEN_IO.INPUT.BUFFER	SCREEN_IO__BUFFER

1.4.2 Order-of-Compilation Rules

The DEC 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 1.4.1.1. 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 4 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 DEC 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 1.4.1.2; what constitutes a complete set of units is discussed in Section 1.4.3; smart recompilation, a compiler feature which significantly reduces the number of compilations needed to rebuild a program after some units change, is discussed in Chapter 5.

1.4.3 Closure

When you compile a given unit, the compiler identifies any unit that the given unit depends on, as specified in Section 1.4.1.1, 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 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.
- 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.

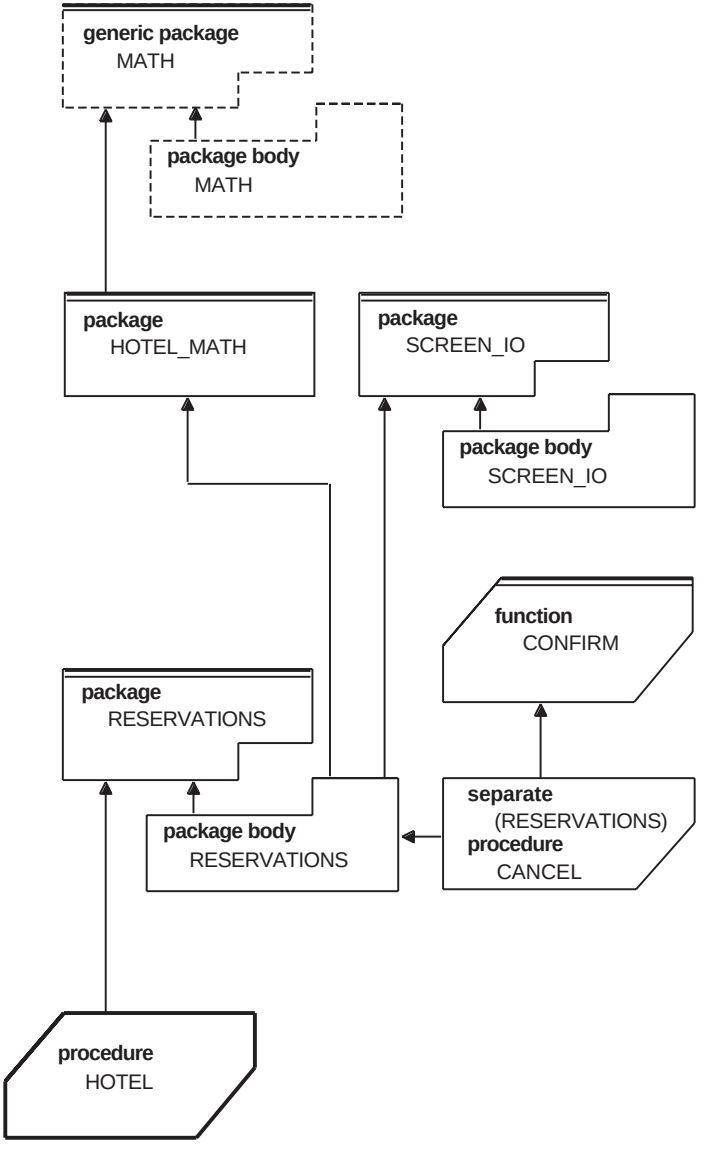
The following command recompiles any of the units shown that are obsolete, except HOTEL (the closure of RESERVATIONS does not include HOTEL):

```
$ 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
```

Figure 1–4 Sample Compilation Units Used to Show Closure



2

Working with DEC 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 DEC Ada *program library* is a dedicated VMS directory that contains a set of files for each compilation unit successfully compiled. A DEC 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 7 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 working with DEC Ada library search paths, see Chapter 3.

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 7 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 directory as a parameter. For example:

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

To create a sublibrary, enter the ACS CREATE SUBLIBRARY command, specifying a 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 7 for more information.

As an alternative to using concealed-device logical names, you can use library search paths. For more information, see Chapter 3.

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

- They create the specified directory (if it does not already exist).
- They cannot be used when a node name is given in a directory specification unless the 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 7 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:
  - FLOAT_REPRESENTATION = VAX_FLOAT
  - LONG_FLOAT = G_FLOAT
  - MEMORY_SIZE = 2147483647
  - SYSTEM_NAME = VAX_VMS or OpenVMS_AXP

  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 7 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 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 current and default paths and system characteristics:

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

Current path evaluates to:

    USER:[JONES.HOTEL.ADALIB]

Program library USER:[JONES.HOTEL.ADALIB]

Created:          10-NOV-1992 14:37:30.63, by DEC Ada V3.0
Last reorganized: <No reorganization date>

Default path in its original form:

    USER:[JONES.HOTEL.ADALIB]

Default path evaluates to:
```

```
    USER:[JONES.HOTEL.ADALIB]

Pragmas that affect STANDARD and SYSTEM:

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

(For information on current and default paths, see Section 3.3.)

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 [SMITH.ADALIB]
Program library USER:[SMITH.ADALIB]

Created:          11-NOV-1992 20:12:47.14, by DEC Ada V3.0
Last reorganized: <No reorganization date>

Default path in its original form:

    USER:[SMITH.ADALIB]

Default path evaluates to:

    USER:[SMITH.ADALIB]

Pragmas that affect STANDARD and SYSTEM:

    pragma FLOAT_REPRESENTATION(VAX_FLOAT)
    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 7 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 7.)

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 the following ACS commands that do not require write access:

- 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 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 (/SUBMIT).

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 if the specified library is accessed via DECnet FAL.

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 directory as a parameter. The directory you specify must be a DEC 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 7 for more information on library protection.

Note that all updates to a program library accessed via the VAX Distributed File System (DFS) is done using the /EXCLUSIVE qualifier. (See the *DECnet for OpenVMS Guide to Networking* for more information on files and networking.)

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 wildcard characters in many ACS commands. The wildcarding rules are similar to those for VMS file specifications (see the *OpenVMS 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 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          11-NOV-1992 17:12:47.41    <entered>

QUEUE
    generic package                11-NOV-1992 17:12:22.54    <entered>
    generic package body           11-NOV-1992 17:12:39.46    <entered>

HOTEL
    procedure body                 11-NOV-1992 11:36:31.26    <main>

SCREEN_IO
    package specification          11-NOV-1992 14:51:03.51
    package body                   11-NOV-1992 17:21:40.01

SCREEN_IO.INPUT
    procedure body                 11-NOV-1992 17:21:51.72
```

```
SCREEN_IO.INPUT.BUFFER
    function body                   11-NOV-1992 17:21:56.00

SCREEN_IO.OUTPUT
    procedure body                  11-NOV-1992 17:21:48.09
Total of 9 units.
```

As shown in the previous 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 default path in its original and evaluated forms (see Chapter 3 for more information)
- 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 name of 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
10-NOV-1992 18:43:29.67

Program library USER:[PROJ]

Created:          10-NOV-1992 14:37:30.63, by DEC Ada V3.0
Last reorganized: <No reorganization date>

Default path in its original form:

    USER:[JONES.HOTEL.ADALIB]

Default path evaluates to:

    USER:[JONES.HOTEL.ADALIB]

Pragmas that affect STANDARD and SYSTEM:

    pragma FLOAT_REPRESENTATION(VAX_FLOAT)
    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:  5-NOV-1992 01:06:13.45
 Entered from: ADA$PREDEFINED_ROOT:[ADALIB]

SCREEN_IO
    Package specification
 Compiled: 10-NOV-1992 18:39:31.09
 Source file: 20-FEB-1992 13:27:03.63
USER:[PROJ]SCREEN_IO_.ADA;7

    Package body
 Compiled: 10-NOV-1992 18:40:57.00
 Source file:  4-NOV-1992 19:39:12.86  USER:[PROJ]SCREEN_IO.ADA;5
```

```
SCREEN_IO.INPUT
    Procedure body
 Compiled: 10-NOV-1992 18:41:14.36
 Source file: 20-FEB-1992 11:56:50.22  USER:[PROJ]SCREEN_IO__INPUT.ADA;3

SCREEN_IO.INPUT.BUFFER
    Function body
 Compiled: 10-NOV-1992 18:41:24.23
 Source file: 20-FEB-1992 11:56:32.72  USER:[PROJ]SCREEN_IO__BUFFER.ADA;3

SCREEN_IO.OUTPUT
    Procedure body
 Compiled: 10-NOV-1992 18:41:30.15
 Source file: 20-FEB-1992 11:57:11.82  USER:[PROJ]SCREEN_IO__OUTPUT.ADA;3

SYSTEM
    Builtin package

TEXT_IO
    Package specification
 Compiled: 10-NOV-1992 18:39:33.73
 Source file: 27-OCT-1992 09:35:43.60  USER:[PROJ]TEXT_IO_.ADA;1

    Package body
 Compiled: 10-NOV-1992 18:39:53.32
 Source file: 27-OCT-1992 09:35:57.80  USER:[PROJ]TEXT_IO.ADA;1
```

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:       11-NOV-1992 14:51:03.51
        Source file:     1-SEP-1991 10:39:54.91  USER:[PROJ]SCREEN_IO_.ADA;2

    Package body
        Compiled:       11-NOV-1992 17:21:40.01
        Source file:     1-SEP-1991 10:39:53.72  USER:[PROJ]SCREEN_IO.ADA;2
        With list:      TEXT_IO
   .
   .
   .
```

You can use the /PORTABILITY qualifier to display a portability summary (see Chapter 7 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:       11-NOV-1992 14:51:03.51
        Source file:     1-SEP-1991 10:39:54.91  USER:[PROJ]SCREEN_IO_.ADA;2

    Package body
        Compiled:       11-NOV-1992 17:21:40.01
        Source file:     1-SEP-1991 10:39:53.72  USER:[PROJ]SCREEN_IO.ADA;2
        With list:      TEXT_IO

   .
   .
   .
PORTABILITY SUMMARY

predefined SHORT_INTEGER or SHORT_SHORT_INTEGER
                        SYSTEM spec

with SYSTEM             TEXT_IO body

predefined floating types in package SYSTEM*
                        TEXT_IO body

enumeration representation clause
                        SYSTEM spec
                        TEXT_IO spec

length SIZE representation clause
                        SYSTEM spec

record representation clause
                        SYSTEM spec

pragma IMPORT_EXCEPTION*
                        IO_EXCEPTIONS spec

pragma IMPORT_FUNCTION* TEXT_IO spec

pragma IMPORT_PROCEDURE*
                        TEXT_IO

pragma INTERFACE        TEXT_IO

pragma INLINE_GENERIC*  TEXT_IO spec

pragma PACK             SYSTEM spec

        where * indicates an implementation-defined feature
```

2.2.4 Checking Unit Currency and Completeness

The DEC 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 syntax-checked units are obsolete:
RESERVATIONS
    package body                    11-NOV-1992 20:38:46.53

%I, The following units depend on missing units:
RESERVATIONS.RESERVE
    procedure body                  11-NOV-1992 17:54:38.98 (00:00:05.50)
RESERVATIONS.RESERVE.BILL
    procedure body                  11-NOV-1992 17:54:45.75 (00:00:05.11)
RESERVATIONS.CANCEL
    procedure body                  11-NOV-1992 17:54:52.07 (00:00:05.28)

1 obsolete unit
```

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, Instantiations within the following units need to be completed (use ACS
COMPILE or ACS RECOMPILE):
GUEST_QUEUE
    package instantiation           20-FEB-1992 16:40:32.84
```

Note that when GUEST_QUEUE is completed, QUEUE_MANAGER, the unit that depends on GUEST_QUEUE, does not become obsolete. See Chapter 4 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 7 for more information on library access and library protection.

As an alternative to copying or entering units, you can use library search paths to share units from other program libraries. For more information, see Chapter 3.

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 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 7 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 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 7 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:

```
$ ACS DIRECTORY
$ STANDARD
package specification          11-NOV-1983 00:00:00.00    <entered>
   .
   .
   .
AUX_IO_EXCEPTIONS
    package specification           3-NOV-1992 11:06:37.81    <entered>

CALENDAR
    package specification           3-NOV-1992 11:27:00.25    <entered>
    package body                    3-NOV-1992 11:27:13.32    <entered>

CDD_TYPES
    package specification           3-NOV-1992 11:27:30.46    <entered>
   .
   .
   .
```

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

An example of entered units is the set of DEC 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. (See Chapter 3 as another way of sharing units across libraries.)

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 the previous 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. For example, by specifying HOTEL as the parameter to the ACS RECOMPILE command, all obsolete units in the closure of HOTEL are recompiled (see Chapter 4 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 DEC 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 DEC Ada import pragma. See Chapter 6 for examples of linking; see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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 the previous example using the DEC 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 6 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          11-NOV-1992 10:11:09.99

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 7 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 current 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%:

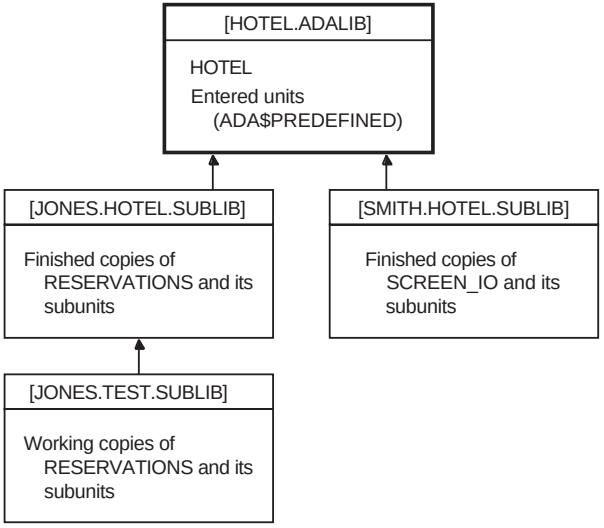
```
$ ACS DIRECTORY A, B*, C%, D
```

2.3.2 Creating a Nested Sublibrary Structure

By using a sublibrary as the parent library of another 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]
```

Figure 2–1 Simple Nested Sublibrary Structure



ZK–6747–GE

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 in the default path of the sublibrary. For example:

```
$ ACS SHOW LIBRARY/FULL [JONES.HOTEL.SUBLIB]
Program library USER:[JONES.HOTEL.SUBLIB]
   .
   .
   .
Default path in its original form:
    USER:[JONES.HOTEL.SUBLIB]
    @USER:[HOTEL.ADALIB]
Default path evaluates to:
    [JONES.HOTEL.SUBLIB]
    USER:[HOTEL.ADALIB]
   .
   .
   .
```

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 7. For general information on concealed-device logical names and rooted directory syntax, see the *OpenVMS User's Manual* and *Guide to OpenVMS File Applications*.

You can also change the parent of a sublibrary by using the ACS MODIFY LIBRARY command to modify the default path of the sublibrary. See Section 3.4 for more information.

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 7 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

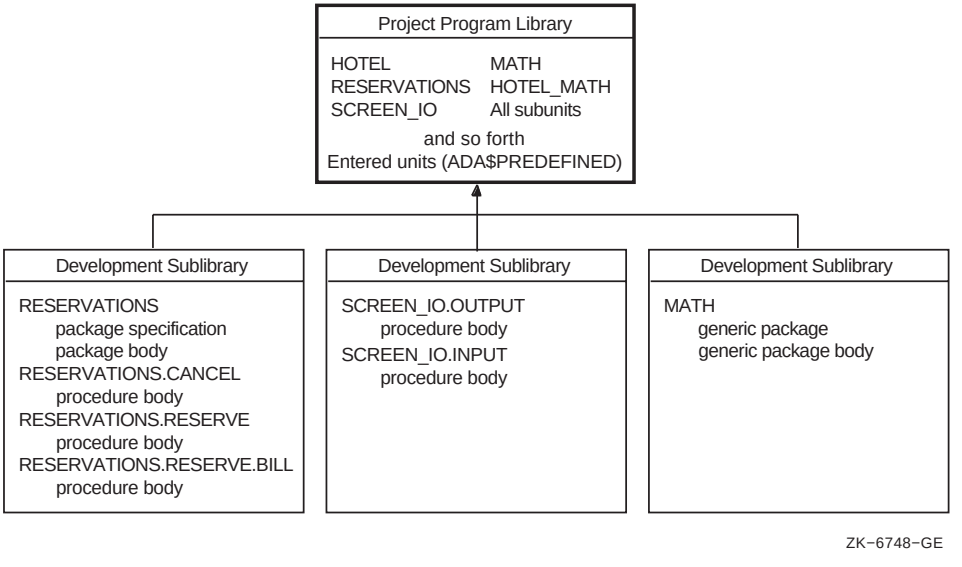
You modify and test units in a sublibrary environment by first isolating the units that need testing. Then, you generally follow these steps:

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 DEC 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 DEC/Code Management System (CMS). See Chapter 7 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.

Figure 2–2 Sublibrary Configuration for the HOTEL Program



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]
```

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          11-NOV-1992 15:16.19.97 (00:00:10)
RESERVATIONS
    package body                   11-NOV-1992 15:15:44.99 (00:00:05)
RESERVATIONS.RESERVE
    procedure body                 11-NOV-1992 15:15:51.63 (00:00:03)
RESERVATIONS.RESERVE.BILL
    procedure body                 11-NOV-1992 15:15:56.79 (00:00:03)
RESERVATIONS.CANCEL
    procedure body                 11-NOV-1992 15:16:02.00 (00:00:04)
```

   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 7 for additional information on configuring sublibrary structures and managing program development.

3

Working with DEC Ada Library Search Paths

In many ways, you can view sublibrary relationships as defining a list of libraries to be searched. When your library context is a sublibrary, the compiler and program library manager search for a unit in the sublibrary first. If the unit is not found, then the search continues in the immediate parent library, the parent of the immediate parent library, and so on. Thus, the search list begins with the sublibrary and follows through the parent libraries until the unit is either found or until there are no more parent libraries to be searched.

DEC Ada library search paths provide another way to define a list of libraries to be searched. Like a sublibrary, a *library search path* simply defines a list of DEC Ada libraries which are searched by the compiler and program library manager to locate units. You can use library search paths to achieve the same effects as sublibraries but with more flexibility.

When you begin development of a program, you may find it convenient to store units in a sublibrary. As your application becomes more complex, a static sublibrary structure may become too unwieldly and difficult to use. Furthermore, you may require that different people work on different components at the same time. In this case, you can use library search paths to add flexibility to your sublibrary structure.

You can use library search paths to dynamically change the relationships among libraries. For example, suppose you want to change the parent of a sublibrary so that you can test units in your sublibrary with another library. You can change the library search path to identify your sublibrary and the other library. (You can achieve the same results using concealed-device logical names and rooted-directory syntax with sublibraries. See Sections 2.3.3 and 7.10.1 for more information.)

You can use library search paths to allow different project members to define different relationships among the same set of program libraries. For example, one project member may be working on one version of the system and another member could be working on a different version. If the units for each version are stored in separate libraries, each member can set up a library search path which identifies the libraries that they need.

Library search paths are implemented in such a way as to be compatible with sublibraries. Furthermore, you can define library search paths to identify independent libraries together with sublibraries.

This chapter explains how you can define, change, and modify library search paths. It also explains how you can use library search paths to configure and reconfigure program libraries.

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 Understanding Current and Default Library Search Paths

A library search path, called the current library search path, or *current path*, defines the particular library search path that is used during compilation or an ACS operation. The compiler and program library manager search for a unit starting with the first library of the current path. If the unit is not found in that library, the search for the unit continues in subsequent libraries in the current path until the unit is either found or until the end of the current path is reached.

For example, suppose that the current path is a list consisting of two libraries: [JONES.SUBLIB] and [JONES.PARENTLIB]. The search for a particular unit begins in the library [JONES.SUBLIB]. If the unit is not found in [JONES.SUBLIB], then [JONES.PARENTLIB] is searched. Note that this search has the same effect as the search from a sublibrary to its parent.

Typically, the current path is a path that is associated with the current program library. This path, called the default library search path, or *default path*, is defined when you create a DEC Ada program library or sublibrary. When you enter the ACS SET LIBRARY command, the specified library becomes the current program library, and the default path associated with that library becomes the current path.

The default path that is defined during library creation differs depending on whether a library or sublibrary is created:

- The default path for a program library is the library itself. With this path, only the library itself is searched.
- The default path for a program sublibrary is the sublibrary itself, followed by the default path of the parent library. With this path, the sublibrary is searched first; then, all libraries in the default path of the parent library are searched in turn.

There are several ways to specify library search paths. The simplest form of a library search path is a list of one or more directory specifications for DEC Ada libraries. For example:

```
[JONES.ADALIB],[HOTEL.ADALIB]
```

You can store library search paths in text files, and then include them in other library search paths. In addition, you can include the default path associated with a library in a library search path. (See Section 3.6 for more information on specifying library search paths.)

Before a unit can be looked up during a library search, the library search path must first be evaluated. During this evaluation, text files and any default paths are included as indicated, and each library in the resulting list is checked to verify that it is a valid DEC Ada library.

When you enter a library search path in a text file or command, the program library manager always saves the path as you typed it without first evaluating it. The current library search path is reevaluated whenever you invoke the compiler or program library manager. (See Section 3.6.1 for more information on how library search paths are evaluated.)

3.2 Defining the Current Path

You can define the current path using one of two methods. In the first method, you enter the ACS SET LIBRARY command. When you enter this command, the specified library becomes the current program library, and the default path associated with that library becomes the current path. For example, suppose that the default path for library [SMITH.SUBLIB] is as follows:

```
[SMITH.SUBLIB]
[HOTEL.ADALIB]
```

In this case, the following command establishes the current program library to be [SMITH.SUBLIB], and the current path to be the default path for [SMITH.SUBLIB]:

```
$ ACS SET LIBRARY [SMITH.SUBLIB]
```

In the previous example, when the program library manager and compiler search for a unit, the library [SMITH.SUBLIB] is searched first. If the unit is not found, then [HOTEL.ADALIB] is searched.

In the second method, you enter the ACS SET LIBRARY command with the /PATH qualifier. This qualifier allows you to establish a current path that is different than the default path associated with a program library. For example, suppose you want to change your current path to be as follows:

```
[SMITH.ADALIB]
[JONES.ADALIB]
[HOTEL.ADALIB]
```

In this case, the following command defines the current path to the desired set of libraries and establishes the current program library to be the first library in the path ([SMITH.ADALIB]):

```
$ ACS SET LIBRARY/PATH [SMITH.ADALIB],[JONES.ADALIB],[HOTEL.ADALIB]
```

In some situations, you may want to identify the default path associated with a library or sublibrary in your current path along with other libraries. To specify a default path, you precede the library with an at sign (@). For example, suppose you want to add the default path of [HOTEL.ADALIB] to [JONES.ADALIB]:

```
$ ACS SET LIBRARY/PATH [JONES.ADALIB],@[HOTEL.ADALIB]
```

In the previous example, the default path of [HOTEL.ADALIB] is included:

```
[HOTEL.ADALIB]
[PROJECT.ADALIB]
```

Thus, the current path evaluates as follows:

```
[JONES.ADALIB]
[HOTEL.ADALIB]
[PROJECT.ADALIB]
```

When you enter the /PATH qualifier, you must specify the at sign (@) to identify the default path associated with a library or sublibrary in the current path. If you do not specify the at sign (@), the Ada compiler and program library manager search for units only in the libraries or sublibraries specified.

For example, the following command establishes the current path to be [SMITH.SUBLIB] (but does not identify any parent libraries):

```
$ ACS SET LIBRARY/PATH [SMITH.SUBLIB]
```

Note that the following commands have the same effect:

```
$ ACS SET LIBRARY [SMITH.SUBLIB]
$ ACS SET LIBRARY/PATH @[SMITH.SUBLIB]
```

Because the ACS SET LIBRARY/PATH command allows you to perform compilations or ACS operations using different libraries or a different library search order than what is specified in the default path, it is useful when you want to temporarily change the current path. If you want to permanently change the default path, use the ACS MODIFY LIBRARY command (see Section 3.4.)

The ACS SET LIBRARY/PATH command stores the specified library search path in the logical name ADA$LIB in the form that you specified. Subsequent invocations of the compiler and program library manager individually reevaluate the current path and reestablish the current program library.

Note that you can define your current program library using the following command:

```
$ ACS SET LIBRARY/PATH @[.ADALIB]
```

In this case, your current program library is always a subdirectory of your current default directory and changes as you change your current default directory.

See Section 3.6 for other ways to specify library search paths.

3.3 Identifying the Current and Default Paths

To identify the current and default paths, enter the ACS SHOW LIBRARY command with the /FULL qualifier. Example 3–1 shows the output of the ACS SHOW LIBRARY/FULL command.

Example 3–1 Output from the ACS SHOW LIBRARY/FULL Command

```
Current program library USER:[JONES.HOTEL.SUBLIB]1

Current path evaluates to:2

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

Program library USER:[JONES.HOTEL.SUBLIB]
Created:          10-NOV-1992 11:35:10.62, by DEC Ada V3.0
Last reorganized: <No reorganization date>
Default path in its original form:3

    USER:[JONES.HOTEL.SUBLIB]
    @USER:[HOTEL.ADALIB]

Default path evaluates to:4

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

   .
   .
   .
```

The following list shows the current and default paths for the library [JONES.HOTEL.SUBLIB]. The numbers match identifying numbers in Example 3–1.

1 The current program library.

2 The current path in its evaluated form. This path is used by the compiler and program library manager to locate units.

3 The default path in the form that you specified. In the previous example, the default path was created when [JONES.HOTEL.ADALIB] and [HOTEL.ADALIB] were created. The at sign (@) which precedes [HOTEL.ADALIB] identifies the default path of that library should be used. In this example, the default path for the library [HOTEL.ADALIB] is itself.

4 The default path in its evaluated form.

If you use the ACS SET LIBRARY/PATH command, the current path in its original form is also listed.

For a library or sublibrary other than the current program library, the ACS SHOW LIBRARY/FULL command displays both the default path in its original form and in its evaluated form.

Note that the current path is not displayed when you use this command for a library other than the current program library.

3.4 Modifying the Default Path

The ACS MODIFY LIBRARY/PATH command redefines the default path of the specified program library. This command stores the new default path in the form that you specified. By default, it also evaluates and verifies the new default path, and reports any errors. For example, to modify the default path of the current program library to also identify the default path of [JONES.SUBLIB], enter the following command:

```
$ ACS MODIFY LIBRARY/PATH [JONES.HOTEL],@[JONES.SUBLIB]
```

In the previous example, the default path is stored exactly as you typed it:

```
[JONES.HOTEL],@[JONES.SUBLIB]
```

Also, the default path of [JONES.SUBLIB] is:

```
[JONES.SUBLIB]
[JONES.PARENTLIB]
```

Thus, the new default path for the library [JONES.HOTEL] evaluates as follows:

```
[JONES.HOTEL]
[JONES.SUBLIB]
[JONES.PARENTLIB]
```

You can use the /EDIT qualifier to invoke an editor to modify the default path. When you specify the /EDIT qualifier, the current definition of the default path is placed in a text file, and the specified editor is invoked. If you do not specify an editor with the /EDIT qualifier, callable EDT is invoked. When you exit from the editor, the default path is redefined to be the library search path contained in the edited file.

When you edit a particular default path for the first time, the term "^__FILE__" is displayed. For more information on this term, see Section 3.6.4.

You can use the /LIBRARY qualifier to modify the default path for a library other than the current program library. For example:

```
$ ACS MODIFY LIBRARY/PATH/LIBRARY=[SMITH.ADALIB] [SMITH.ADALIB],@[JONES.HOTEL]
```

In the previous example, the default path of [SMITH.ADALIB] has been modified as follows:

```
[SMITH.ADALIB]
@[JONES.HOTEL]
```

Suppose the default path of [JONES.HOTEL] is as follows:

```
[JONES.SUBLIB]
[JONES.PARENTLIB]
```

In this situation, the new default path for user SMITH evaluates as follows:

```
[SMITH.ADALIB]
[JONES.HOTEL]
[JONES.SUBLIB]
[JONES.PARENTLIB]
```

You can also specify the ACS MODIFY LIBRARY/PATH command with the /[NO]VERIFY qualifier. When you specify the /NOVERIFY qualifier, the program library manager suppresses the evaluation and verification of the new default path. The default for this qualifier is /VERIFY.

If you enter the ACS MODIFY LIBRARY/PATH command interactively (at the ACS> prompt), the current path is not reevaluated. Thus, if you modified the default path for the current program library, this modification will not take effect until you invoke the Ada compiler or reinvoke ACS.

3.5 Configuring and Reconfiguring Program Libraries Using Library Search Paths

During the different phases of development, you may want to configure or reconfigure your program library for several reasons:

- To work on a different part of the program
- To work with more stable or less stable versions of code
- To allow others to work on an individual programmer's code

DEC Ada library search paths allow you to configure and reconfigure your program library to suit the needs of your project.

Figure 3–1 shows a program library configuration for the user Smith. Note the following points about Figure 3–1:

- The library [SMITH.ADALIB] is Smith's private program library. Smith is developing a part of the HOTEL program called SCREEN_IO.

- Smith's current program library is [SMITH.ADALIB]. The current path, which is indicated by the arrows, is as follows:
  - The library [SMITH.ADALIB]
  - The library [PROJECT.ADALIB]
- The library [PROJECT.ADALIB] is the project library that contains stable versions of the HOTEL program. The default path associated with this library is the library itself.
- The library [JONES.ADALIB] is Jones' private program library. Jones is developing another part of the HOTEL program called RESERVATIONS.
- Jones' current program library is [JONES.ADALIB]. The current path, which is not indicated by the arrows, is as follows:
  - The library [JONES.ADALIB]
  - The library [PROJECT.ADALIB]

Suppose that Smith wants to create a temporary library configuration that allows testing of changes to SCREEN_IO with Jones' changes to RESERVATIONS. For this configuration, Smith wants the library search order to be:

- Smith's library ([SMITH.ADALIB])
- Jones' library ([JONES.ADALIB])
- Project library ([PROJECT.ADALIB])

One way to reconfigure Smith's program library is to modify the default path associated with the library [SMITH.ADALIB] using the ACS MODIFY LIBRARY command. The problem with modifying a default path is that the change affects all library search paths that reference the default path for that library.

A better way to reconfigure Smith's program library is to redefine the current path to the desired configuration. By using the ACS SET LIBRARY/PATH command, Smith can test Jones' code without changing any of the default paths associated with the libraries in the reconfiguration. For example, Smith can redefine the current path to identify Smith's library ([SMITH.ADALIB]) and the default path of Jones's library (@[JONES.ADALIB]):

```
$ ACS SET LIBRARY/PATH [SMITH.ADALIB],@[JONES.ADALIB]
```

Figure 3–1 Program Library Configuration for Smith

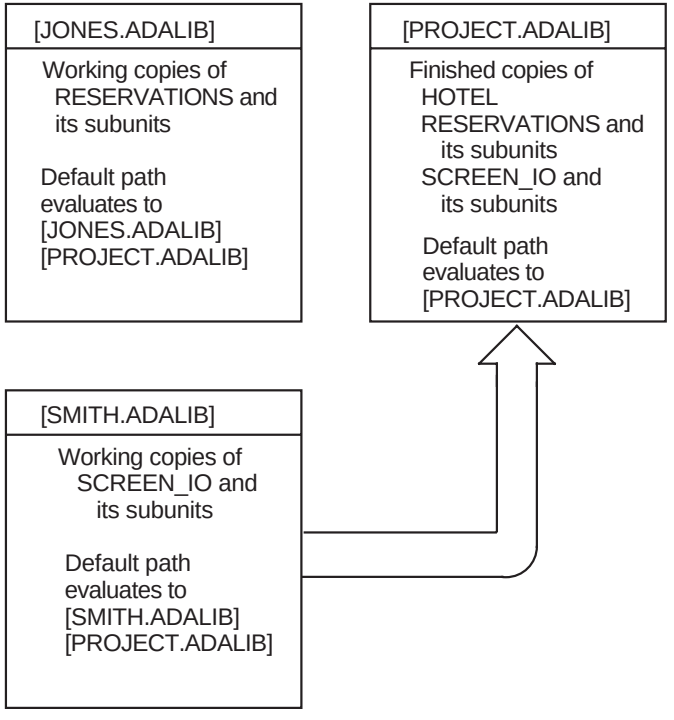
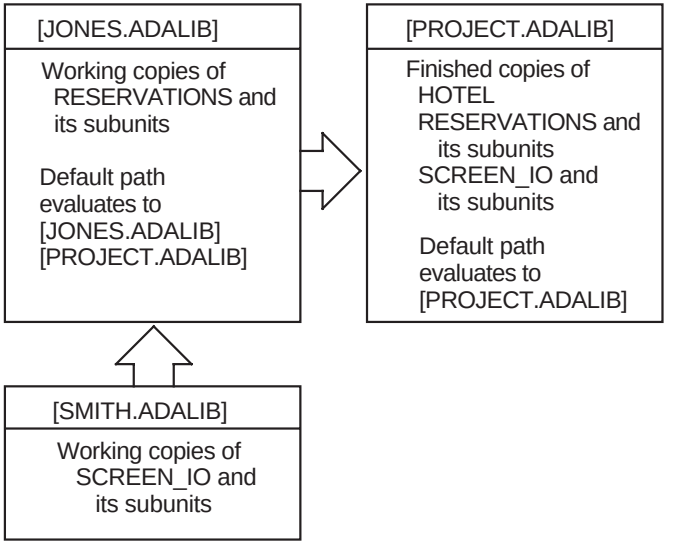


Figure 3–2 shows the reconfigured library structure.

Once testing is completed, Smith can change back to the previous configuration as follows:

```
$ ACS SET LIBRARY [SMITH.ADALIB]
```

Figure 3–2 Program Library Reconfiguration for Smith



ZK–4677A–GE

3.6 Specifying Library Search Paths

You specify library search paths as a expression consisting of a list of terms. A term in a path expression can be:

- A directory name for an Ada library
- A reference to the default path for an Ada library
- A reference to a library search path stored in a text file

Expressions in library search paths must be evaluated before a unit can be looked up during a library search. The following sections describe how library search paths are evaluated and provide detailed information on specifying library search paths in commands, in files, and as a default path.

3.6.1 Understanding How Library Search Paths are Evaluated

When you enter a DEC Ada command, the terms in the current path are evaluated to form an ordered list of DEC Ada program libraries. Table 3–1 shows what terms can appear in a path expression and the result of their evaluation.

Table 3–1 Results of Evaluating Terms in Path Expressions

Term	Description	Result of Evaluation
dirname	A directory name of an Ada library	The value of *dirname* is appended to the ordered list of Ada libraries that define a library search path.
@dirname	Directory name of an Ada library preceded by @	The default path associated with a library is evaluated.
@file-spec	File specification preceded by @	The path expression in the file is evaluated. If you do not specify a full file specification, the default file name is PATH and the default file extension is .TXT.

As the terms in path expressions are evaluated, the names of directories that result from the evaluation are appended to the resulting list in the order in which these expressions are specified. If a program library occurs more than once in the resulting list of libraries, subsequent occurrences are ignored by the compiler and program library manager.

Library search paths are not allowed to cycle. In other words, if the library search path for library [ALIB] contains [BLIB], then the library search path for library [BLIB] cannot contain library [ALIB] in its library search path.

3.6.2 Specifying Library Search Paths in Commands

You can specify library search paths with several commands:

- As a command parameter to the ACS SET LIBRARY/PATH command
- As a command parameter to the ACS MODIFY LIBRARY/PATH command
- As the value of the /LIBRARY qualifier when you also specify the /PATH qualifier with the DCL ADA command

When you specify library search paths in ACS commands, note the following point:

- You must use commas to separate terms in path expressions. For example:

```
$ ACS SET LIBRARY/PATH [JONES.ADALIB], [HOTEL.ADALIB]
```

When you specify library search paths in the DCL ADA command, note the following point:

- If a term in the value of the /LIBRARY qualifier contains the at sign (@), you must use double quotes (") to surround that term. For example:

  ```
  $ ADA/LIBRARY=([JONES.ADALIB], "@[HOTEL.ADALIB]")/PATH
  ```

  Note that you can only specify a library search path as a value to the /LIBRARY qualifier when you also specify the /PATH qualifier.

3.6.3 Specifying Library Search Paths in Files

You can store library search paths in text files, and then include the files in other library search paths by using the at sign (@). Terms in a library search path in a text file are separated by commas, spaces, or carriage returns.

For example, the file MYPATH.TXT is a text file containing the following libraries:

```
[JONES.HOTEL.ADALIB]
[HOTEL.ADALIB]
```

When the following command is followed by a subsequent compilation or ACS operation, the libraries [JONES.TEST.SUBLIB], [JONES.HOTEL.ADALIB], and [HOTEL.ADALIB] are searched in order for the required units.

```
$ ACS SET LIBRARY/PATH [JONES.TEST.SUBLIB], @MYPATH.TXT
```

Two restrictions apply when you store a library search path in a file:

- The maximum length of a line in the file is 256 characters.
- The at sign (@) must appear on the same line as the directory or file specification that follows it.

3.6.4 Specifying Library Search Paths in Default Paths

A special notation is used in the default path of a library to refer to that library in a symbolic manner so that you can move the library to another directory without having to modify the default path.

The special notation used for such symbolic self-references is the string ^__FILE__. (The nonalphabetic characters in the string are a circumflex and a double underscore.) When a default path for a library is evaluated, ^__FILE__ is replaced by the directory name of that library.

For example, suppose you modified the default path of [JONES.TEST.SUBLIB] to be as follows:

```
^__FILE__
[JONES.HOTEL.ADALIB]
```

When this path is evaluated, ^__FILE__ is replaced with the directory name of an Ada library ([JONES.TEST.SUBLIB]). Thus, the default path of the library [JONES.TEST.SUBLIB] evaluates as follows:

```
[JONES.TEST.SUBLIB]
[JONES.HOTEL.ADALIB]
```

Suppose you used the backup utility to move all the files in [JONES.TEST.SUBLIB] to [SMITH.TEST.SUBLIB]. When the default path of [SMITH.TEST.SUBLIB] is evaluated, ^__FILE__ is replaced by [SMITH.TEST.SUBLIB]. Thus, the resulting default path of the library [SMITH.TEST.SUBLIB] is as follows:

```
[SMITH.TEST.SUBLIB]
[JONES.HOTEL.ADALIB]
```

4

Compiling and Recompiling DEC Ada Programs

In DEC 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.

By default, whenever a unit is compiled, any dependent units may become 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.

DEC 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 4–1 summarizes and compares the characteristics and use of each command.

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

Table 4–1 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 4–1 (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 form of diagnostic messages is described in Appendix E.

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.

Many examples in this manual were created with smart recompilation in effect. For more information about smart recompilation, see Chapter 5.

4.1 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 4–2.

Table 4–2 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 interactive 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 4.2).
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 the previous 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, Invoking the DEC Ada compiler
```

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 4.2 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 *OpenVMS Debugger Manual*).

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.

4.2 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 FLOAT_REPRESENTATION, 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 4.1).
- One or more units were loaded into the library using the ACS LOAD/DESIGN command. These units are design-checked only (see Appendix D for more information on program design support).

To recompile a set of obsolete units, you can enter either the ACS RECOMPILE or the ACS COMPILE command. In DEC 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 4–3 notes the differences between the ACS RECOMPILE and COMPILE commands in performing these steps.

Table 4–3 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 are potentially obsolete, as follows:

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

The following RECOMPILE command operates on the closure of RESERVATIONS, and compiles units that become obsolete by the change to the specification of SCREEN_IO.

```
$ ACS RECOMPILE/LOG RESERVATIONS
%I, The following syntax-checked units are obsolete:
SCREEN_IO
    package specification           13-NOV-1992 12:10:17.54

%I, The following units may also be recompiled:
SCREEN_IO.INPUT
    procedure body                  13-NOV-1992 11:39:11.67 (00:00:06.42)
SCREEN_IO.INPUT.BUFFER
    function body                   13-NOV-1992 11:39:19.18 (00:00:04.10)
SCREEN_IO.OUTPUT
    procedure body                  13-NOV-1992 11:39:24.21 (00:00:04.61)
RESERVATIONS
    package body                    13-NOV-1992 11:39:35.04 (00:00:10.61)
    package specification           13-NOV-1992 11:38:55.34 (00:00:04.30)
RESERVATIONS.RESERVE
    procedure body                  13-NOV-1992 11:39:46.79 (00:00:05.34)
RESERVATIONS.RESERVE.BILL
    procedure body                  13-NOV-1992 11:39:53.27 (00:00:04.79)
RESERVATIONS.CANCEL
    procedure body                  13-NOV-1992 11:39:59.39 (00:00:05.64)

1 obsolete unit, 7 possibly obsolete (total 8)

%I, Invoking the DEC Ada compiler
```

Note that when smart recompilation is in effect, only those units that are actually affected by the change are recompiled. See Chapter 5 for more information.

As shown in the previous example, you can use the /LOG qualifier to display the potentially obsolete units and the order in which they may be recompiled. Obsolete units that are actually affected by the changes are recompiled using the copied source files in the current program library.

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. By default (/PRELOAD), the ACS COMPILE command compiles a modified set of units whose compilation order has changed (or to which new units have been added) in the correct order (see Section 4.4).

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 previous 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
```

4.3 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 may become 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 may also need to be recompiled.

See the *DEC Ada Language Reference Manual* and *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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-1992 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-1992 16:35

$ ACS RECOMPILE/LOG ACCOUNTING
%I, The following units will be completed:
ACCOUNTING
    package instantiation          15-Apr-1992 16:35

$ 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;
```

Suppose 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-1992 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-1992 21:15
    Corresponds to generic package GENERIC_PACKAGE compiled
        8-Mar-1992 21:16
 .
 .
 .
```

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

4.4 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 4.6. 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 4.2 for more information on how the COMPILE command recompiles obsolete units. See Section 4.3 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 the previous 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 syntax-checked units are obsolete:
RESERVATIONS
    package specification          15-DEC-1992 17:27:32.94
    package body                   15-DEC-1992 17:27:29.90
RESERVATIONS.RESERVE
    procedure body                 15-DEC-1992 17:21:37.33
RESERVATIONS.RESERVE.BILL
    procedure body                 15-DEC-1992 17:21:34.41
RESERVATIONS.CANCEL
    procedure body                 15-DEC-1992 17:21:36.08

5 obsolete units

%ACS-I-CL_COMPILING, Invoking the DEC Ada compiler

5 units compiled in 00:00:45
```

By default, the ACS COMPILE command (by way of the /PRELOAD qualifier) looks within a source file to determine the use of **with** clauses, subunit stubs, and so on when it does a compilation. Thus, this qualifier allows 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).

4.5 Forcing the Recompilation of a Set of Units

In some cases, you may want to force the recompilation of a set of units. For example, you may want to recompile a set of units with different qualifiers, such as /NOOPTIMIZE or /NOCHECK. Because the ACS COMPILE and RECOMPILE commands do not recompile current units by default, you can use the /OBSOLETE qualifier to accomplish this task.

The /OBSOLETE qualifier allows you to specify a set of units to be considered obsolete when determining the currency of the units in the closure. For example, the following command specifies that the specification and body of RESERVATIONS should be considered obsolete and forces their recompilation (and possibly the recompilation of any dependent units) with the /NOOPTIMIZE qualifier:

```
$ ACS RECOMPILE/OBSOLETE=UNIT:RESERVATIONS/NOOPTIMIZE RESERVATIONS
```

You can use wildcards to specify units in the /OBSOLETE=UNIT:unit-name qualifier. For example, to force the recompilation of the entire closure with the /NOCHECK qualifier, enter the following command:

```
$ ACS RECOMPILE/OBSOLETE=UNIT:*/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 /OBSOLETE=BODY:unit-name qualifier if a unit is current, but you want to force the recompilation of only its body. 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 considers obsolete and forces the recompilation of the body (and possibly any dependents of the body) of SCREEN_IO:

```
$ ACS RECOMPILE/OBSOLETE=BODY:SCREEN_IO RESERVATIONS
```

Note that you cannot recompile entered units using the ACS COMPILE or RECOMPILE commands.

4.6 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 CMS library. See Chapter 7 for more information on the interaction between CMS and the DEC 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
```

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

4.8 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:

- Executing the ACS LOAD, COMPILE, or RECOMPILE compilations in a subprocess (the default mode).
- Batch processing. When processing in batch mode, use a dedicated batch queue 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.
- 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.

4.8.1 Loading Units and Executing Compilations in a Subprocess

By default (/WAIT), the ACS LOAD, COMPILE, or RECOMPILE commands are executed in a subprocess.

The following example creates a subprocess and invokes the compiler command file created by the program library manager to load the closure of unit RESERVATIONS:

```
$ ACS LOAD 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.

4.8.2 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 the *DEC Ada Installation Guide for OpenVMS VAX Systems*. 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 *OpenVMS DCL Dictionary* for more information on these options.

You can also submit ACS LOAD, COMPILE, and RECOMPILE compilation in batch mode. To execute these compilations in batch mode, you must use the /SUBMIT qualifier with these commands. The ACS 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 DEC 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.

4.8.3 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/NOOPTIMIZE SCREEN_IO
  $ ACS COMPILE/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.

4.8.4 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

When you specify the /SUBMIT qualifier, 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.

4.8.5 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 *n*th 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.

5

Using the Professional Development Option

DEC Ada provides several features and capabilities which support the development of large Ada programs. These features and capabilities are licensed separately under the Professional Development option.

The Professional Development option includes the following features and capabilities:

- Smart recompilation

  This feature can significantly reduce the number of recompilations that are needed to rebuild your program after some compilation units change. Smart recompilation allows the compiler to propagate changes quickly through a system, eliminating up to 100% of the usual recompilations. (See Section 5.1 for more information.)

- Program Library File-Block Caching

  This feature minimizes the actual amount of disk input-output that must be performed by using an in-memory cache of file blocks from the .ACU files. As a result of file-block caching, the elapsed time for compilations is significantly reduced. (See Section 5.2 for more information.)

- Directory Structure

  This feature improves the performance of access to large program libraries. (See Section 5.3 for more information.)

All of the features and capabilities of the Professional Development option are designed so that you do not need to change your current development procedures, source code, or program libraries when you first enable it. Once the license is enabled, you can use your currently existing programs and program libraries without any special linkers, loaders, or conversion procedures.

Note that if you do not have a license for the Professional Development option, the features and capabilities discussed in this chapter are not available for use. (For information on how to obtain a license for the Professional Development option, contact your local Digital sales representative.)

The following sections describe the Professional Development option features and capabilities.

5.1 Overview of Smart Recompilation

Smart recompilation is a compiler feature that reduces the number of compilations that are needed to rebuild a program after some compilation units change. When you compile a unit whose source file has changed, only those dependent units affected by the changes are recompiled, rather than all of the dependent units.

When smart recompilation is in effect, the compiler stores dependence information at a more detailed level than it normally does. This information describes the dependences of a unit at a finer level than the compilation unit level. During a subsequent compilation, the compiler uses this information to determine which units are actually affected by a source code change and recompiles only those units affected by the change. (For more information about dependences, see Section 5.1.5.)

When smart recompilation is not in effect, a unit is obsolete when any of the units it depends on have been compiled more recently than it. Because that unit is obsolete, all units that depend on it are also obsolete and need to be recompiled. However, because smart recompilation handles unit dependences at a much finer granularity, it prevents this domino effect of recompiling obsolete units just because they depend on a unit that has been compiled.

To invoke smart recompilation, use the /SMART_RECOMPILATION qualifier with the DCL ADA and the ACS CHECK, COMPILE, LOAD, RECOMPILE, and SHOW PROGRAM commands. When the license for Professional Development option is enabled, /SMART_RECOMPILATION is the default for these commands. (See the *OpenVMS License Management Utility Manual* for more information on enabling licenses.)

When smart recompilation is in effect, note the following points:

- You can use entered units in the same way as you use them without smart recompilation. Specifically, reentering a unit into the program library after recompiling it in its original library has the same effect as recompiling it into the local library. In other words, units that depend on the entered unit become obsolete only if they depend on fragments of the entered unit that changed.

- The compiler uses the same rules as described in Chapters 2 and 3 when searching for units and determining where the results of the compilation should be stored. Specifically, any obsolete compilation units are recompiled into the program library (and current units are not updated into the program library).

5.1.1 Using Smart Recompilation to Recompile Obsolete Units

During incremental program development, you may need to rebuild your program frequently. For example, you may need to rebuild your program after you a make change to one or more units, add new units, or recompile parts of your program with new qualifiers.

When smart recompilation is in effect, modified source files are always compiled. However, the ACS COMPILE and RECOMPILE commands use the detailed dependence information to detect when an unmodified unit in the closure is unaffected by changes (if any) in the units it depends on. The ACS COMPILE and RECOMPILE commands do not recompile such dependent units and thus minimize unnecessary recompilations.

Conversely, when smart recompilation is not in effect—that is, you explicitly specify the /NOSMART_RECOMPILATION qualifier—units are considered obsolete and are recompiled based on their time of compilation. In addition, if the /NOSMART_RECOMPILATION qualifier is specified, detailed information about dependences is not stored in the program library.

The following example shows the use of the RECOMPILE command with smart recompilation. 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.

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

- The body of SCREEN_IO.
- The subunits of SCREEN_IO.
- The specification of RESERVATIONS is potentially affected because it names SCREEN_IO in a **with** clause.
- The body of RESERVATIONS is potentially affected because it depends on the specification of RESERVATIONS.

- The subunits of RESERVATIONS are potentially affected because they depend on the body of RESERVATIONS.
- The unit HOTEL is potentially affected because it names the unit RESERVATIONS in a **with** clause.

When you enter the ACS RECOMPILE command, it uses the detailed information about dependences stored in the program library, and may determine that only the body of RESERVATIONS needs to be recompiled. Further, the ACS RECOMPILE command determines the currency of the units that are dependent on the body of RESERVATIONS (and any subunits of RESERVATIONS) only after the body of RESERVATIONS is recompiled. For example (because the /SMART_RECOMPILATION qualifier is the default, you do not need to specify it on the command line):

```
$ ACS RECOMPILE/LOG HOTEL
%I, The following units will be recompiled:
RESERVATIONS
    package body                    16-DEC-1992 12:47:54.60 (00:00:08.61)

%I, The following units may also be recompiled:
RESERVATIONS.RESERVE
    procedure body                  16-DEC-1992 12:44:22.50 (00:00:01.48)
RESERVATIONS.RESERVE.BILL
    procedure body                  16-DEC-1992 12:44:24.61 (00:00:01.28)
RESERVATIONS.CANCEL
    procedure body                  16-DEC-1992 12:44:26.54 (00:00:01.50)

1 obsolete unit, 3 possibly obsolete (total 4)
Total elapsed time for last compilation of all 4 units was 0:00:12.87

%I, Invoking the DEC Ada compiler
%I, Package body RESERVATIONS added to library
        Replaces older version compiled 16-DEC-1992 12:47:54.60
   .
   .
   .
1 unit compiled in 00:00:09, 3 units did not need to be recompiled
Estimated elapsed time savings due to Smart Recompilation was 0:00:04.26 (33%)
```

As shown in the previous example, the /LOG qualifier displays the units that will be recompiled and the units that may be recompiled.

Note that when smart recompilation is not in effect, all units that are directly or indirectly dependent on the specification of SCREEN_IO are recompiled. For example:

```
$ ACS RECOMPILE/LOG/NOSMART_RECOMPILATION HOTEL
I, The following units will be recompiled:
RESERVATIONS
    package specification          16-DEC-1992 12:44:15.17 (00:00:01.88)
    package body                   16-DEC-1992 12:51:50.26 (00:00:08.35)
HOTEL
    procedure body                 16-DEC-1992 12:44:17.69 (00:00:01.66)
RESERVATIONS.RESERVE
    procedure body                 16-DEC-1992 12:44:22.50 (00:00:01.48)
RESERVATIONS.RESERVE.BILL
    procedure body                 16-DEC-1992 12:44:24.61 (00:00:01.28)
RESERVATIONS.CANCEL
    procedure body                 16-DEC-1992 12:44:26.54 (00:00:01.50)
SCREEN_IO
    package body                   16-DEC-1992 12:44:28.72 (00:00:00.99)
SCREEN_IO.INPUT
    procedure body                 16-DEC-1992 12:44:34.08 (00:00:07.09)
SCREEN_IO.INPUT.BUFFER
    function body                  16-DEC-1992 12:44:42.05 (00:00:01.27)
SCREEN_IO.OUTPUT
    procedure body                 16-DEC-1992 12:44:43.93 (00:00:00.88)

10 obsolete units
Total elapsed time for last compilation of the 10 units was 0:00:26.38

%I, Invoking the DEC Ada compiler
   .
   .
   .
10 units compiled in 00:00:39
```

5.1.2 Determining the Impact of a Change

During program development, you may want to determine the impact of a change without actually compiling the change into the program library. To determine the impact of a change, follow these steps:

1. Create a sublibrary that has as its ancestors the desired program libraries. (Alternatively, you can create a program library and include the desired libraries in your path. See Chapter 3.) For example:

   ```
   $ ACS CREATE SUBLIBRARY/PARENT=[PROJECT.ADALIB] [.TMPLIB]
   ```

2. Define the newly created sublibrary as the current program library using the ACS SET LIBRARY command. For example:

   ```
   $ ACS SET LIBRARY [.TMPLIB]
   ```

3. Make the desired changes to your Ada source files and compile them into the library. For example, if you modified the body of the unit RESERVATIONS, you enter the following command to compile that unit into your program library:

```
$ ADA RESERVATIONS
```

4. Enter the ACS CHECK command to determine the impact of the change. For example, you can determine the impact of a change to the unit RESERVATIONS on the HOTEL program by entering the following command:

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

%I, The following units need to be recompiled:
RESERVATIONS.RESERVE
    procedure body                  16-DEC-1992 13:14:02.08 (00:00:01.61)

%I, The following units may also need to be recompiled:
RESERVATIONS.RESERVE.BILL
    procedure body                  16-DEC-1992 13:06:19.55 (00:00:01.40)

1 obsolete unit, 1 possibly obsolete (total 2)
Total elapsed time for last compilation of all 2 units was 0:00:03.01
```

   In this example, the ACS CHECK command lists the units that are affected by the change to RESERVATIONS, but it does not actually compile or recompile these units.

   Alternatively, you can enter the ACS SHOW PROGRAM command to obtain the same information in a different format. In addition, the ACS SHOW PROGRAM command provides a detailed listing of all of the units (obsolete and current) in the closure and lists information about the program library.

5.1.3 Forcing Recompilation when Smart Recompilation is in Effect

When smart recompilation is in effect, you may want to force a unit or a set of units to be recompiled that would not otherwise be recompiled. For example, you may want to recompile a set of units with different qualifiers. Because smart recompilation does not recompile dependent units if they are not affected by a change, you must explicitly force the recompilation of those units.

To force the recompilation of a set of units, you can enter either of the following commands:

- ACS RECOMPILE/OBSOLETE=UNIT:*unit-name*
- ACS RECOMPILE/NOSMART_RECOMPILATION

You enter the ACS RECOMPILE/OBSOLETE=UNIT:*unit-name* command when you want to force the recompilation of the specified unit only. Dependent units are not recompiled because the specified unit is not actually obsolete (neither its source nor the units it depends on have changed). For example, the following command forces the unit RESERVATIONS to be recompiled:

```
$ ACS RECOMPILE/OBSOLETE=UNIT:RESERVATIONS RESERVATIONS
```

Note that you can force the recompilation of the unit RESERVATIONS and all of its dependents with the /OBSOLETE=UNIT:* qualifier. For example:

```
$ ACS RECOMPILE/OBSOLETE=UNIT:* RESERVATIONS
```

A less desirable way to force recompilation of the specified unit and all of its dependent units is to use the ACS RECOMPILE/NOSMART_RECOMPILATION command. Using the /NOSMART_RECOMPILATION qualifier has the following effects:

- Obsolesence is determined by using the compilation date.
- Detailed information about dependences is not created for the units specified.

For more information on forcing the recompilation of units, see Section 4.5.

5.1.4 Optimizing the Development Environment for Smart Recompilation

You can optimize the development environment in the following ways:

- Use library search paths and sublibraries whenever possible.

  To avoid the risk of having the main library being potentially unusable while an unpredictable number of recompilations happen, use library search paths to perform compilations into a sublibrary. Once the compilations have finished, compilation units can be merged into the main parent library.

  For example, this is especially useful when using a Ram disk, such as DECRam, as there is no reason to ever write the temporary sublibrary to a physical disk.

- Force the recompilation of units in your library periodically using the ACS RECOMPILE/OBSOLETE command.

  You can achieve better compile-time performance by periodically forcing recompilation. When you periodically force the recompilation of units, the detailed information is updated so that dates can be used to check for obsolescence in subsequent compilations. (Obsolescence checks between two units is more efficient if a date check (rather than a check through the

detailed information) is used.) See Section 5.1.3 for more information on forcing recompilation.

5.1.5 Understanding Inter-Unit Dependences

The following sections describe inter-unit dependences as they relate to some Ada language features and smart recompilation.

5.1.6 Fragments, Inter-Dependence, and Independence

In general, smart recompilation breaks the compilation unit into fragments, where each fragment is either a complete declaration or part of one. The inter-compilation-unit dependences are established on one or more of these fragments. When the information in one of these fragments changes, the compilation units that depended on it become obsolete.

Smart recompilation uses the following rules when establishing inter-unit dependences:

- A fragment in a package specification or body does not include other fragments of the specification or body. For example, a call to a procedure body does not become obsolete because you changed the statements within a procedure, unless that procedure is inlined. Similarly, if you add, remove, or change declarations inside a package, smart recompilation does not make the references to other declarations within the package obsolete.
- A fragment for a variable usually does not include its initial expression. A fragment for a constant object is similar but may include the initial expression if the expression is a compile-time constant or constrains the object.

  This means that changes to most initial expressions will not cause compilation units to become obsolete.
- The fragment for an incomplete or private type is merged with the fragment for the full type if the two fragments are in the same compilation unit. Thus, adding components to record types in private parts does make uses of the private type obsolete.
- Changes in one fragment will make another fragment obsolete only if the other fragment makes a reference to the first, either directly or indirectly. (Reducing such interactions is one of the reasons that initial expressions are not included in variable fragments.)

Note that changes to comments has no effect on fragments, and therefore, does not cause any dependent units to be recompiled.

5.1.6.1 Searching for Identifiers and Overloading Resolution

When smart recompilation is in effect, dependences are created between units when identifiers are searched. Dependences are created when the identifiers are found and when they are not found.

Inter-unit dependences start when one unit names another unit in a **with** clause. Most references into the unit within the **with** clause now take place either as part of the compiling of a selected name (for example, TEXT_IO.NEW_LINE) or as a search of a series of scopes for a simple name (for example, **use** TEXT_IO;...NEW_LINE;...).

Searching for a name can be viewed as a search of a series of declaration lists for a specific identifier or operator. Smart recompilation records each scope and the name being sought.

When the entity is found, a dependence is made on the corresponding fragment.

Note that *not* finding an occurrence of the identifier or operator is just as important as finding it. If a future version of the scope has this identifier added to it, then this search would be affected, and so the unit whose compilation caused this search needs to become obsolete.

This means that adding a new nonoverloadable declaration (for example, a type, subtype, constant, variable, or generic) will usually not cause compilation units to become obsolete. However, a **use** clause or uplevel reference from a subunit may have caused this declaration list to be searched for this identifier, so the searching compilation unit must now be obsolete.

Adding overloadable declarations such as subprograms or enumerals will similarly cause all compilation units that searched this declaration list for the added identifier to become obsolete.

Adding a new nonlimited type also adds the =, /= operations for that type, and perhaps other operators, so all compilation units that searched this declaration list for one of these will become obsolete.

Similarly, when a type is modified (for example, by adding another component), the type changes. This means that all the subprograms that have a parameter or result of this type change. Therefore, all compilation units that searched this declaration list for one of these identifiers become obsolete because overloading resolution might behave differently.

5.1.6.2 Resolving Access Types

There are a few other ways that dependences are established. For example, resolving an allocator (X := **new** INTEGER;) requires an exhaustive search for access types whose accessed type is INTEGER.

Adding new access types may thus result in compilation units containing such allocators to become obsolete.

5.1.6.3 Inlining and Generic Expansion

All generic instantiations depend on all of the generic specification, and any reordering or changing the contents of this generic specification will make the compilation unit containing the instantiation obsolete. However this is not too harmful because of the way the instantiation is processed. The instantiation produces a specification which is treated like a normal specification. Each declaration in the resulting specification is matched against the old instantiation, and usually ends up being a compatible replacement for it, so the dependents do not become obsolete.

All inlined subprogram calls and inlined generic instantiations depend on all of the body, and any reordering or changing of the contents of the body makes the compilation unit containing the call or instantiation obsolete. Again, recompiling this compilation unit will usually not make its dependents obsolete.

Note

On the common case that dependent units do not need to be recompiled, the previous source files for the replaced generic unit or inlined subprogram should be retained in the debugger source list. This enables source line debugging of the calls or instantiations to work, since the dependents units' debugging information refers to these previous source files.

5.1.6.4 With and Use Clauses

Changing **with** clauses is like adding or removing a declaration in the package STANDARD. The body or subunit compilation units of this unit become obsolete if they searched the package STANDARD for this identifier. Similarly, any body or subunit that is affected by a **use** clause in a specification or parent compilation unit becomes obsolete when that **use** clause no longer applies. Adding a **use** clause can cause new scopes to be searched which might contain declarations that are homographs of previously resolved declarations. This could cause declarations no longer to be visible (see the *DEC Ada Language*

Reference Manual). Therefore, a body or subunit is considered obsolete if a **use** clause is added to a specification or parent.

5.1.6.5 Pragmas and Representation Clauses

In general, changing or adding a representation clause or pragma for a declaration will make all the compilation units that depended on the declaration obsolete.

In general, changing the pragma INTERFACE or any related import-export pragma invalidates the declaration these pragmas apply to, and invalidates all other declarations of the same identifier in the same declaration list.

Changing the pragma ELABORATE may make a compilation unit's dependents obsolete, especially if they have exploited the presence of the pragma to eliminate an access-before-elaboration check.

5.1.7 Coding Your Programs to Use Smart Recompilation Efficiently

In general, you do not need to change your coding conventions to use smart recompilation. However, there are some conventions that are good to follow. For example:

- Use **use** clauses carefully.

  Use clauses increase the number of declaration lists searched for identifiers, so you should not use them indiscriminately. Placing them in block statements around the few statements where they are convenient is better than placing them in outer scopes. This placement limits the number of identifiers that are looked up in the declaration lists, and reduces the dangers of a clash when more declarations are added. This is especially true for such subprograms as + and =, where the addition of a type in one of the used packages will make all the lookups of the operators obsolete.

- Do not place unrelated type declarations in the same package specification.

  Placing unrelated type declarations in the same package specification increases the number of declaration lists that are searched. When coding unrelated type declarations, use nested subpackages to group types and operations. Each additional declaration list reduces the likelihood of changes in identifier searches, and thereby reduces the likelihood of compilation units becoming obsolete.

- Implement widely occurring types that change frequently as access types to an incomplete type. For example:

```
package FOO_SUPPORT is
   type FOO is limited private;
   procedure GET_C1 (F : in out FOO; Value : Integer);
   procedure SET_C1 (F : in out FOO; Value : Integer);
   procedure GET_C2 (F : in out FOO; Value : Integer);
   procedure SET_C2 (F : in out FOO; Value : Integer);
   procedure COPY (From : FOO; To : in out FOO);
   procedure FINALIZE (F : in out FOO);
private
   type FOO_INFO;
   type FOO is access FOO_INFO;
end;

package body FOO_SUPPORT is
    type FOO_INFO is record ...
    end record;
    .
    .
    .
end;
```

  The style in the previous example has both benefits and costs as follows:

  – A benefit is that adding new components to the full type FOO_INFO in the package body and adding the corresponding GET*n* and SET*n* subprograms in the package specification will not make any dependent packages obsolete. This is because the subprograms being added do not have the same identifiers as existing subprograms, and thus smart recompilation avoids recompiling the dependents of FOO_SUPPORT.

  – The biggest development cost is the run-time cost in allocating and deallocating objects of the type FOO and a development cost in the extra lines of code required to do the deallocation.

  – Another cost, the performance of calling the GET and SET subprograms, can be eliminated near the end of the project by adding the pragma INLINE or INLINE_GENERIC. However, adding these pragmas too early during development causes extra dependences and recompilations.

5.2 Overview of Program Library File-Block Caching

One of the features provided by the Professional Development option is program library file-block caching. This feature is enabled when the license is enabled, and you do not need to specify any qualifiers or define any logicals to cause file caching to occur.

Whenever the compiler is invoked, compilation units and the units they depend on (directly and indirectly) are accessed by reading the .ACU files in the program library. These files are accessed in an arbitrary manner such that not all blocks are read, and some blocks are read multiple times. The order in which the .ACU files are read depends on the following:

- The particular unit dependences
- The phase of the compiler making use of the information represented by the dependence

To minimize the actual amount of disk input-output that must be performed, the DEC Ada compiler contains an in-memory cache of file blocks from the .ACU files. When the compiler needs a particular block, it first looks in the cache and uses the data there. If the block is not in the cache, then a single disk read is performed to bring the block and several adjacent blocks disk. This may cause other blocks to be displaced from memory if the cache is not large enough to hold all of the blocks needed during the compilation.

The program library file-block caching feature can significantly reduce the elapsed time for compilation as follows:

- When the block is already in the cache, there is no elapsed-time expense.
- When the block must come from disk storage, several blocks are accessed in a single operation so the single disk read latency is amortized over several blocks.
- When you compile a single unit, these benefits apply because the compiler may need to read the same block several times to access unrelated data during different compiler phases.

The Professional Development option optimizes the file-block cache for large programs. The cache size is adjusted to increase the probability of cache hits (to reduce the number of useful blocks that are displaced from the cache).

The major benefits accrue when multiple units are compiled at once, either using the DCL ADA or the ACS COMPILE or RECOMPILE command. Typically, the units being compiled depend on a common set of .ACU files; these files are read into memory when the first unit is compiled and used from memory for the subsequent units.

5.3 Overview of the Directory Structure Feature

With the Professional Development option, the program libary manager reduces the time required to perform ACS operations for large programs. This time savings is achieved by employing multiple subdirectories in the program library. Without the Professional Development option, a DEC Ada program library is implemented as a dedicated directory that contains a set of files for each unit compiled into the program library. For a program library that contains a large number of compilation units, the program library directory contains a correspondingly large number of files and the directory (.DIR) file for the program library directory also becomes correspondingly large.

When a directory file becomes larger than 128 blocks, access to files becomes inefficient, particularly when the files are being created or deleted. Therefore, access to the library files in large program libraries can become slow.

To improve the performance of access to large program libraries, the Professional Development option causes the program library manager to create a series of subdirectories within the program library. Then, whenever units are added, changed, or deleted from a program library, they are updated in the subdirectories rather than in the library directory itself. Since these operations are performed on relatively small directories, access to the program library remains fast even for large program libraries. Furthermore, the program library manager adjusts the number and size of the subdirectories as needed.

Note that program library manager creates subdirectories automatically once the license for the Professional Development option is enabled.

DEC Ada supports program libraries created or updated with and without the Professional Development option in effect. Specifically, DEC Ada without the Professional Development option supports program libraries in which the library files exist in library subdirectories rather than in the library directory, and DEC Ada with Professional Development option supports program libraries in which the library files exist in the library directory rather than in the library subdirectories.

6
Linking Programs

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

DEC Ada programs are linked using the VMS Linker. To link DEC Ada object modules, you invoke the linker through the program library manager using the ACS LINK command (you do not invoke the 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 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 6.3 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 DEC 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 DEC Ada environment.

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

6.1 Linking Programs Having Only DEC Ada Units

If your program consists only of DEC 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 4 for more information on recompiling obsolete units and completing units containing incomplete generic instantiations.

6.2 Linking Mixed-Language Programs

The DEC 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 DEC 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.

6.2.1 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 DEC 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 the preceding 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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* and *DEC Ada Language Reference Manual* for detailed information on the syntax for and use of the DEC 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.

6.2.2 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 DEC Ada main program:

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

In DEC 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 DEC 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 DEC 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, DEC 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
```

6.2.3 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 DEC 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 *DEC Ada Language Reference Manual* and *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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 previous 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 7 and Appendix A for more information.

6.3 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 4).
- 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 *OpenVMS Linker Utility Manual*.

6.3.1 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
  ```

Note that some new qualifiers are available with the linker that are not supported by the ACS LINK command. You can pass such qualifiers to the linker by defining a symbol like the following before invoking the ACS LINK command:

```
$ LINK :== LINK/NATIVE_ONLY/SYSEXE
```

This symbol is then used by the ACS LINK command procedure that invokes the linker.

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

By default (/WAIT), the link operation for the ACS LINK command is executed in a subprocess. 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 4).

6.3.3 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.

7

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 DEC Ada.

7.1 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 4 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 *DEC Ada Language Reference Manual* for more information on packages, generic units, and subunits. See Chapter 1 for more information on unit dependences.

Example 7–1 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 7–1 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 7–1 (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;
```

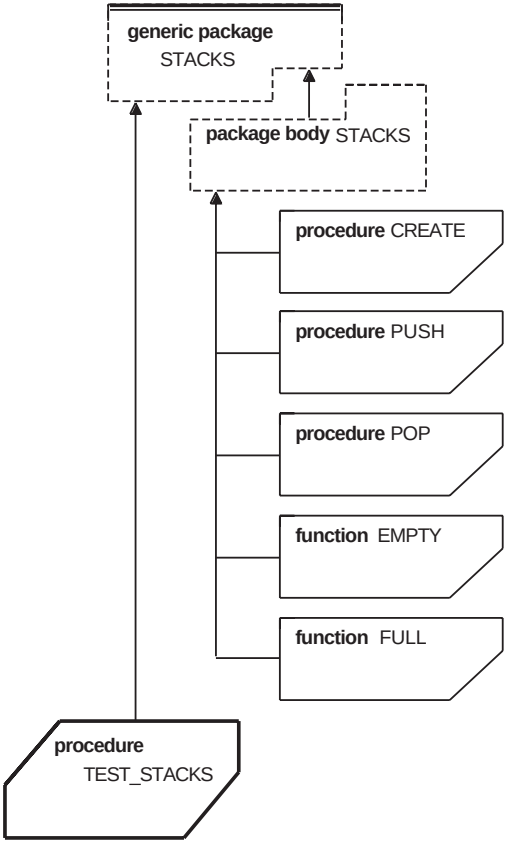
(continued on next page)

Example 7–1 (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 7–1 diagrams the application in Example 7–1 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 4 for more information on incomplete units, obsolete units, and generic completions.

Figure 7–1 Diagram of Decomposed Stack Application



7.2 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 7–2 shows a library structure for the decomposed stack application from Example 7–1. Note the following points about Figure 7–2:

- 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 7–2 does not show multiple versions of the application, but additional sublibraries could be used to create and develop different versions or other development streams. Furthermore, you can use library search paths to create new relationships among program libraries (see Chapter 3).

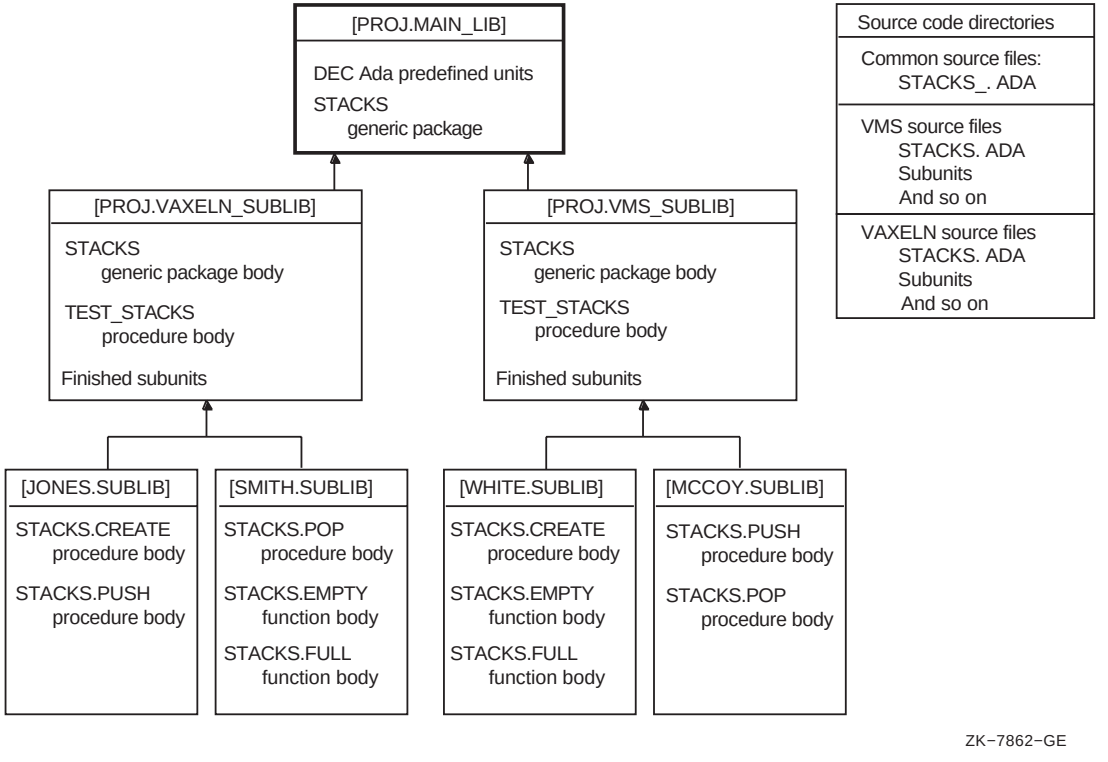
A structure like the one in Figure 7–2 allows testing from the bottom up. See Chapter 2 for additional information on developing and testing units in sublibraries.

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 7–2 shows one source code directory to the right of the program library structure. See Section 7.3.1 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.
- You can store the stable units in a separate library and define your current path to identify that library plus any other libraries that you need.

Figure 7–2 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

7.3 Integration with Other DEC Tools

Like other DEC languages and layered products, DEC 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 DEC Ada to manage program development:

- DEC/Code Management System (CMS)
- Language-Sensitive Editor (LSE)

For general information on creating a software environment, see *Using DECset for VMS Systems*. This manual describes how to create a development environment using the DEC Software Engineering Tools (DECset). DECset includes LSE, SCA, CMS, as well as the Module Management System (MMS), Test Manager, and the Performance and Coverage Analyzer (PCA).

7.3.1 Setting up Source Code Directories

An effective way to set up and manage source code directories is to use CMS. The *Guide to Code Management System for VMS Systems* and *Using DECset for VMS Systems* give detailed information on how to use CMS.

You can use CMS libraries in conjunction with DEC 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 DEC Ada program libraries. Or, you can divide up your source code among several CMS libraries that are associated with one or more DEC Ada program libraries.

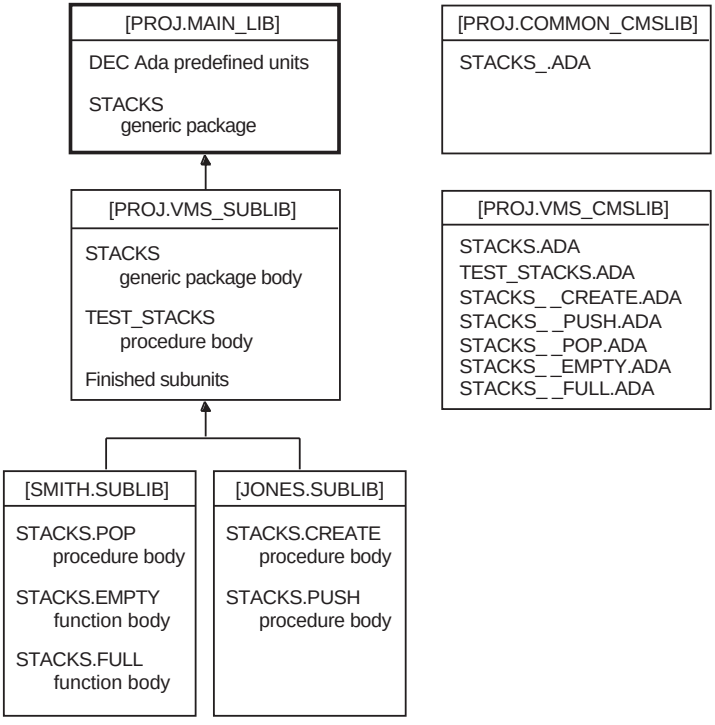
Beginning with Version 3.0, CMS allows you to use search lists to manage multiple libraries. So, you can construct trees of CMS libraries that parallel your DEC Ada program libraries and sublibraries. Figure 7–3 shows one such configuration.

The following search list applies to the library structure in Figure 7–3:

```
$ 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 DEC Ada sublibraries for compilations, and you can use it for retrieving and modifying source code in

Figure 7–3 Ada Program Library and Sublibrary Structure with CMS Libraries



ZK–7861–GE

the same way that you use DEC Ada sublibraries to test Ada compilations (see Chapter 2).

7.3.2 Managing Source Code Modifications

LSE, CMS, and the DEC 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 Language-Sensitive Editor for VMS Systems* 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 7–2 sends all Ada compilations to whatever queue is represented by the logical name ADA$BATCH.

Example 7–2 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
$ 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/CLOSURE TEST_STACKS
```

Because the ACS COMPILE command assumes the /PRELOAD qualifier by default, you can compile a modified set of units whose compilation order has changed in the correct order. However, if you have created new source files to add new units to your program library, you must add the units to 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 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 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.

7.4 Efficient Use of DEC Ada on VMS Systems

The impact of Ada compilations on the performance on VMS systems can be minimized in the following ways:

- Plan to minimize the size of your compilations by decomposing your applications and structuring your libraries and sublibraries as shown in Section 7.1.
- Use a suitable 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.

For more information, see *DEC Ada Installation Guide for OpenVMS VAX Systems* or *DEC Ada Installation Guide for OpenVMS AXP Systems*.

7.4.1 Reducing Disk Traffic Times

Disk traffic elapsed times are usually much more seriously affected by the number of disk-head movements required for reading and writing, rather than the exact amount of information read or written. This means that you can reduce disk traffic times if you are careful about how much data you store and retrieve on a regular basis. The following discusses some way that you can reduce the amount of data you store thereby reducing disk traffic elapsed time.

- 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 file and copied source file. DEC Ada is careful not to store unnecessary information in the program library. In particular, the compiler prunes as much information as possible from the .ACU files and stores these files in a compact format.

By default (/COPY_SOURCE), the DEC Ada compiler keeps copies of the source files for each unit compiled. Copied source files can be used later for recompilations and debugging.

However, you can reduce the amount of data you store in your program library if you use the /NOCOPY_SOURCE qualifier with the DCL ADA, and ACS COMPILE and RECOMPILE commands. If you use this qualifier and you subsequently compile or debug your program, the source file from the original library from which you compiled the unit will be used.

Note that if you move your source files after you have compiled them into a program library using the /NOCOPY_SOURCE qualifier, a subsequent recompile or invocation of the debugger will not find the original source files.

- Although the CMS allows you to compile programs directly from sources in CMS libraries, CPU time and disk traffic generated from doing so is much higher than going directly through RMS to the source file. For this reason, using CMS reference copy directories or some equivalent technique is recommended.
- The actual amount of disk space consumed by a project reflects the size of the project and the number of copies of the compilation units kept. Sublibraries and search paths can be used to reduce the amount of disk space consumed, however, each library in the search path must be searched in turn until a unit is found.
- DEC Ada performs better if you invoke the program library manager interactively rather than in the form of one-line DCL commands.

  The direct inout-output count for the example showing ACS commands in the form of one-line DCL commands is almost double that of the interactive one. Executing ACS interactively takes advantage of caching within ACS, the Ada compiler, and RMS. If you execute ACS commands noninteractively, the caching is lost when the image is rundown and reactivated.

  You can also reduce disk input-output if you enter several units with a single invocation of an ACS command. For example:

```
$ ACS DIRECTORY HOTEL,RESERVATIONS*
```

- The DEC Ada and program library manager rely on system caches to reduce disk input-output traffic. Compiling and recompiling units using ACS COMPILE and RECOMPILE commands interactively uses the caches more efficiently.

7.4.2 Reorganizing Library Structures

The DEC Ada library structure is an ISAM file, and as such benefits from being correctly tuned. The ACS REORGANIZE command does this tuning, and should be used especially for improving the performance of large libraries that are frequently updated.

7.5 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 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 access control lists (ACLs) on program libraries for each kind of library access

For complete details on file and directory protection, see the *OpenVMS VAX Guide to System Security* or the *OpenVMS AXP Guide to System Security*, *OpenVMS DCL Dictionary*, and *VMS Access Control List Editor Manual*.

7.5.1 Program Library Access Requirements for ACS Commands

The program library manager recognizes three kinds of program library access (not to be confused with 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 directory associated with the library)

Table 7–1 lists the kinds of access required by each of the ACS commands.

Table 7–1 Program Library Access Needed to Use ACS Commands

ACS Command	Library Access	Comments
CHECK	R	
COMPILE	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.
MODIFY LIBRARY	RW	
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	
SET LIBRARY/EXCLUSIVE	RW	Exclusive access is also needed.

(continued on next page)

Table 7–1 (Cont.) Program Library Access Needed to Use ACS Commands

ACS Command	Library Access	Comments
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.

7.5.2 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 the following 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 7.5.1, 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 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.

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 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 7–2 shows how the UIC-based protection for each file in a program library is related to the program library access discussed in Section 7.5.1. Table 7–2 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 7–2 Minimum UIC Protection for Each Kind of Library Access

Program Library Access (see Section 7.5.1)	Library Index File and Library Version Control File (UIC Access)	Other Library Files (UIC Access)	Directory File (UIC Access)
R	R	R	R
RW	RW	RWD	RW
RWD	RWD	RWD	RWD[1]

[1]If the 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 7–2, library index file UIC protection must be the same as the directory file protection. To ensure this, you can use the /PROTECTION qualifier when you create the library. However, if the 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 7–2 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 Appendix A.

7.5.3 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 7.5.2 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 *OpenVMS VAX Guide to System Security* or the *OpenVMS AXP Guide to 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 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 7–1).

As Table 7–2 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 *OpenVMS VAX Guide to System Security* or the *OpenVMS AXP Guide to 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 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 directory associated with the program library, according to the information given in Table 7–2. Normally, you should not need to do this; keep in mind that putting an excessive number of ACLs on all files in the program library will result in performance penalties for both users of the DEC 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 *OpenVMS VAX Guide to System Security* or the *OpenVMS AXP Guide to System Security* for more information on access request evaluation.

7.6 Accessing Program Libraries from Multiple Systems

For certain development projects, the project program libraries may be set up on multiple computer systems. In order to allow access to these libraries from other systems, note the following points:

- Use VAXclusters, whenever possible.

  When the program library is on a VAXcluster, the disk containing the library should be mounted on all nodes requiring access to the library. All compiler and program library functionality is available on a VAXcluster.

- If you cannot use VAXclusters, consider using VAX Distributed File System (DFS) on VAX systems.

  DFS provides a low-overhead network-based file system over DECnet, and can be used in either local area network (LAN) or wide-area network (WAN) configurations. DFS does not support shared-write access to disks accessed over the network. Access to DFS-mounted disks is performed with an implicit /EXCLUSIVE qualifier. Futhermore, all accesses to the program library share a single DECnet link. See Sections 7.7 and 7.8 for guidelines on configuring DFS-based program libraries.

- DEC Ada supports access using the DECnet File Access Listener (FAL). FAL allows a library to be on any VMS node accessible via DECnet. All of the compiler and program library manager functions are available, with the following exceptions:

  – The program library is not supported for /EXCLUSIVE access.

  – You cannot create a program library or sublibrary using DECnet FAL, unless the directory already exists.

  Because each file accessed using DECnet FAL will require a separate DECnet link, this may impose considerably more load upon the network, and on the system hosting the library, than a DFS-based solution.

7.7 General Guidelines for Network Access

When accessing program libraries over the network, you can use either DECnet FAL or DFS. The following sections provide some guidelines for accessing libraries over the network.

Note that if you are using DEC Ada in a cluster environment, the following sections on accessing program libraries using DFS and FAL do not apply.

7.7.1 Network Protection Mechanisms for Program Libraries

When a project requires network access to a program library, it is necessary to ensure that the protection mechanisms in place are appropriate.

In some cases, it may make sense to make an entire library world-readable, and let users access it using default DECNET accounts. For example, a library containing public utility packages can be protected such that the library is world-readable. In other cases, project members may require write-access to a library on a different, non-clustered node. For this type of access, each project member is granted a default proxy into either their own account or group account on the node hosting the library. See Section 7.5.1 for information.

Note that the proxies granted must be default proxies, regardless of whether DFS or FAL is used. See the *OpenVMS System Manager's Manual* for more information on how to create proxy accounts.

7.7.2 Achieving Efficient Network 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 over the network. Your system manager can help you with this or you can consult the *OpenVMS System Manager's Manual* for more information.

7.7.3 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 7.10.5 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.

7.8 Accessing Program Libraries Using DFS

The following sections provide some guidelines for specifying program libraries using DFS.

7.8.1 Configuring a Program Library using DFS

Since DFS does not allow shared-write access to a file, it is generally advisable to have your working sublibrary on a local disk and any parent libraries and sublibraries on either a local or DFS-mounted disk. If the working sublibrary is on a DFS-mounted disk, accessing the sublibrary from multiple processes (such as an interactive process or a batch stream), may fail.

Consider the configuration decribed in Figure 7–3. In that example, the sublibrary [SMITH.SUBLIB] should be located on a disk which is local to the machine that SMITH normally uses; the sublibrary [PROJ.VMS_SUBLIB] and the library [PROJ.MAIN_LIB] can be on disks which are either local or DFS-mounted onto that machine.

If DFS is running over a slow or overloaded link, it may be advisible to use a cache sublibrary as suggested in Figure 7–4. However, if the library is accessible over a fast link (such as FDDI or moderately loaded ethernet), the cache library may be unnecessary.

7.9 Accessing Progam Libraries with DECnet FAL

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.

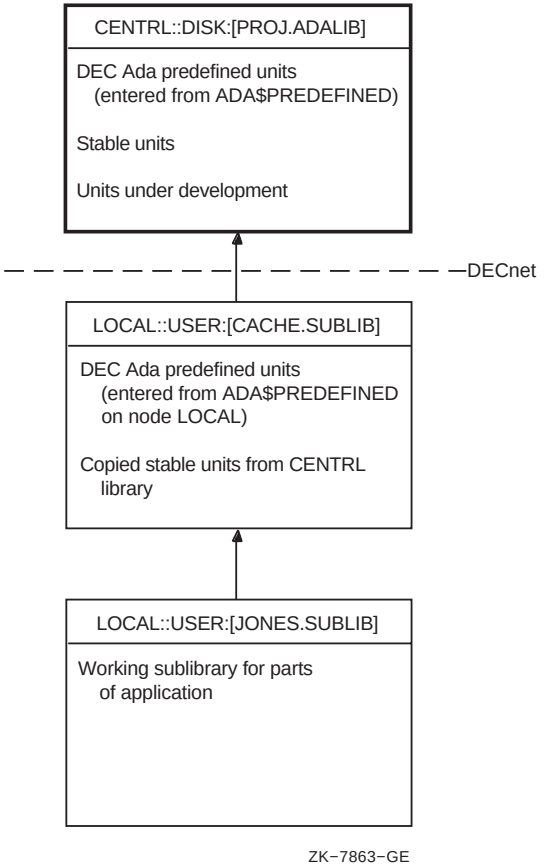
The following sections give some guidelines for specifying program libraries using DECnet FAL. In particular, Section 7.9.2 lists any restrictions that may apply.

For more information on DECnet, see the *DECnet for OpenVMS Guide to Networking*.

7.9.1 Configuring a Library Structure using DECnet FAL

If you plan to configure a system using DECnet FAL, 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 using DECnet FAL. Figure 7–4 suggests one such configuration and consists of the following libraries:

Figure 7–4 DECnet Program Library Configuration



- A 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.

When working with libraries which are accessed using DECnet FAL, be sure to consider system security. For maximum security, use proxy accounts (see the *DECnet for OpenVMS Guide to Networking* and the *DECnet for OpenVMS 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. Otherwise, if the program library is accessed using DECnet FAL, access links can accumulate when you perform parent library operations.

7.9.2 Restrictions on Using Program Libraries Accessed by DECnet FAL

Observe the following restrictions when distributing program libraries which are accessed using DECnet FAL:

- In the absence of the VAX Distributed File System (DFS), CMS (Version 3.0 and lower) does not support access using DECnet FAL. Thus, the program library manager may issue an error if an operation requires accessing a CMS library using DECnet FAL.
- Directories cannot be created using DECnet FAL. Thus, the ACS CREATE LIBRARY and CREATE SUBLIBRARY commands can be used to create program libraries or sublibraries across the network unless the corresponding 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.

7.10 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 DEC 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.

7.10.1 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 7.10.1.1 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 7.10.1.2 discusses rooted directories.

You can make logical name assignments at the system, group, or job level, as appropriate. The *DEC Ada Installation Guide for OpenVMS VAX Systems* and the *DEC Ada Installation Guide for OpenVMS AXP Systems* 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 *OpenVMS DCL Dictionary* and the *Guide to OpenVMS File Applications*.

As an alternative to using concealed-device logical names, you can use library search paths to dynamically create library relationships. See Chapter 3 for more information.

7.10.1.1 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 DEC 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.

7.10.1.2 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]
```

7.10.2 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 backup utility (see Section 7.10.3).

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[,...]
   .
   .
   .
```

7.10.3 Backing Up and Restoring Program Libraries

To back up program libraries, use the 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 saveset on node CENTRL:

```
$ BACKUP [PROJ.JONES]HOTEL_ADALIB.BCK/SAVE_SET -
_$ [PROJ.JONES.ADALIB]
```

See the *OpenVMS System Manager's Manual* 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 7.10.1.1 and 7.10.1.2 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.

7.10.4 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 the previous 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.

7.10.5 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 7.5.1 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 the previous 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 previous 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 the previous 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.

7.10.6 Recompiling Units After a New Release or Update of DEC Ada

When an update or full release of DEC 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 DEC Ada predefined units from the current program library:

   ```
   $ ACS REENTER *
   ```

   Consult the cover letter and release notes supplied with the release of DEC Ada that you are using. If new units have been added to the DEC 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 *
   ```

7.11 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.

7.11.1 Determining DEC Ada Program Portability

To determine if your DEC 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 DEC 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 DEC Ada program, and identify those features that may appear in the portability summary.

7.11.1.1 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 DEC 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 DEC Ada—which may or may not be significant to your program.

The DEC 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 DEC 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 DEC 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.

7.11.1.2 Features Listed in the Portability Summary

The portability summary lists one or more of the features or constructs shown in Table 7–3. 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 7–3 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 7–3 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)*
predefined LONG_INTEGER

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 integer types in the package SYSTEM*
predefined floating point types in the package SYSTEM*
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 function IMPORT_VALUE*
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 7–3 (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 7–3 (Cont.) Features or Constructs that May Appear in a Portability Summary

Pragmas
unknown pragmas (*that is, any pragma not recognized by DEC Ada*)
unsupported pragmas (*that is, any pragma supported by another Ada implementation*)
pragma AST_ENTRY*
pragma COMMON_OBJECT*
pragma COMPONENT_ALIGNMENT*
pragma EXPORT_EXCEPTION*
pragma EXPORT_FUNCTION*
pragma EXPORT_OBJECT*
pragma EXPORT_PROCEDURE*
pragma EXPORT_VALUED_PROCEDURE*
pragma FLOAT_REPRESENTATION*
pragma IDENT*
pragma IMPORT_EXCEPTION*
pragma IMPORT_FUNCTION*
pragma IMPORT_OBJECT*
pragma IMPORT_PROCEDURE*
pragma IMPORT_VALUED_PROCEDURE*
pragma INTERFACE
pragma INTERFACE_NAME*
pragma INLINE_GENERIC*
pragma LONG_FLOAT*
pragma MAIN_STORAGE*
pragma MEMORY_SIZE
pragma PACK
pragma PRIORITY
pragma PSECT_OBJECT
pragma SHARED

(continued on next page)

Table 7–3 (Cont.) Features or Constructs that May Appear in a Portability Summary

Pragmas
pragma SHARE_GENERIC*
pragma STORAGE_UNIT
pragma SUPPRESS
pragma SUPPRESS_ALL*
pragma SYNCHRONIZE*
pragma SYSTEM_NAME
pragma TASK_STORAGE*
pragma TIME_SLICE*
pragma TITLE*
pragma VOLATILE*

7.11.2 Setting the System Name

The DEC Ada program library manager, as the interface to the DEC Ada compiler and linker, is sensitive to differences in targets through the value of the predefined constant SYSTEM_NAME in the package SYSTEM. On VMS systems, this constant can have a value of either OpenVMS_AXP, 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 an OpenVMS_AXP, 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 on VAX systems or OpenVMS_AXP on AXP systems. 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,
  ACS SET PRAGMA/SYSTEM_NAME=OpenVMS_AXP, 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 6 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 OpenVMS_AXP, 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.

8

Debugging DEC Ada Tasks

All of the debugger techniques covered in the *OpenVMS Debugger Manual* 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 *DEC Ada Language Reference Manual* and *DEC Ada Run-Time Reference Manual for OpenVMS Systems*. (See Appendix C for supplemental information on debugging DEC Ada programs.

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.

8.1 A Sample Tasking Program

Example 8–1 demonstrates a number of common errors that you may encounter when debugging tasking programs. The calls to procedure BREAK 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 call to the procedure BREAK and display the current state of each task:

```
DBG> SET BREAK %LINE  46 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  71 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  76 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  92 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE 100 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE 104 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE 120 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 CHILD. This task is declared by task FATHER, and is designated %TASK 3 in the SHOW TASK displays.
- A single task named MOTHER. This task is declared by the main program, and is designated %TASK 4 in the SHOW TASK displays.

Example 8–1 Procedure TASK_EXAMPLE

```
 1 package TASK_EXAMPLE_PKG is
 2    procedure BREAK;
 3 end;
 4
 5 package body TASK_EXAMPLE_PKG is
 6    procedure BREAK is
 7    begin
 8       null;
 9    end;
10 end;
```

(continued on next page)

Example 8–1 (Cont.) Procedure TASK_EXAMPLE

```
11
12
13 with TEXT_IO; use TEXT_IO;
14 with TASK_EXAMPLE_PKG; use TASK_EXAMPLE_PKG;
15 procedure TASK_EXAMPLE is 1
16
17    pragma TIME_SLICE(0.0); -- Disable time slicing. 2
18
19    task type FATHER_TYPE is
20       entry START;
21       entry RENDEZVOUS;
22       entry BOGUS; -- Never accepted, caller deadlocks.
23    end FATHER_TYPE;
24
25    FATHER : FATHER_TYPE; 3
26
27    task body FATHER_TYPE is
28       SOME_ERROR : exception;
29
30       task CHILD is  4
31          entry E;
32       end CHILD;
33
34       task body CHILD is
35       begin
36          FATHER_TYPE.BOGUS;   -- Deadlocks on call to its parent
37       end CHILD;              -- (parent does not have an accept
38                               -- statement for entry BOGUS). Whenever
39                               -- a task-type name (here, FATHER_TYPE)
40                               -- is used within a task body, the
41                               -- name designates the task currently
42                               -- executing the body.
43    begin -- (of FATHER_TYPE body)
44
45       accept START do
46          BREAK;   -- Main program is waiting for this rendezvous to
47                   -- complete; CHILD is suspended when it calls the
48                   -- entry BOGUS.
49          null;
50       end START;
51
```

(continued on next page)

Example 8–1 (Cont.) Procedure TASK_EXAMPLE

```
52       PUT_LINE("FATHER is now active and"); 5
53       PUT_LINE("is going to rendezvous with main program.");
54
55       for I in 1..2 loop
56          select
57             accept RENDEZVOUS do
58                PUT_LINE("FATHER now in rendezvous with main program");
59             end RENDEZVOUS;
60          or
61             terminate;
62          end select;
63
64          if I = 2 then
65             raise SOME_ERROR;
66          end if;
67       end loop;
68
69    exception
70       when OTHERS =>
71          BREAK;   -- CHILD is suspended on entry call to BOGUS.
72                   -- Main program is going to delay while FATHER
73                   -- terminates.
74                   -- MOTHER is ready to begin executing.
75          abort CHILD;
76          BREAK;   -- CHILD is now abnormal due to the abort statement.
77
78          raise; -- SOME_ERROR exception terminates FATHER.
79    end FATHER_TYPE;
80
81 begin    -- (of TASK_EXAMPLE)  6
82
83    declare
84       task MOTHER is  7
85          entry START;
86          pragma PRIORITY (6);
87       end MOTHER;
88
89       task body MOTHER is
90       begin
91          accept START;
92          BREAK;   -- At this point, the main program is waiting for
93                   -- its dependents (FATHER and MOTHER) to terminate.
94                   -- FATHER is terminated.
95          null;
96       end MOTHER;
```

(continued on next page)

Example 8–1 (Cont.) Procedure TASK_EXAMPLE

```
 97    begin  8
 98
 99
100       BREAK;   -- FATHER is suspended at accept start.
101                -- CHILD is suspended in its deadlock.
102                -- MOTHER has activated and ready to begin executing.
103       FATHER.START;  9
104       BREAK;   -- FATHER is suspended at its 'select or terminate'
105                -- statement.
106
107
108       FATHER.RENDEZVOUS;
109       FATHER.RENDEZVOUS;  10
110       loop  11
111          -- This loop causes the main program to busy wait
112          -- for the termination of FATHER, so that FATHER
113          -- can be observed in its terminated state.
114          if FATHER'TERMINATED then
115             exit;
116          end if;
117          delay 1.0;
118       end loop;
119
120       BREAK;   -- FATHER has terminated by now with the unhandled
121                -- exception SOME_ERROR. CHILD no longer exists
122                -- because its master (FATHER) has terminated. Task
123                -- MOTHER is ready.
124       MOTHER.START; 12
125          -- The main program enters a wait-for-dependents state,
126          -- so that MOTHER can finish executing.
127    end;
128 end TASK_EXAMPLE; 13
```

Key to Example 8–1:

1 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 16 through 80).

2 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 (On VAX systems, time slicing is disabled if the pragma TIME_SLICE is omitted or is specified with a value of 0.0; On AXP systems, pragma TIME_SLICE (0.0) must be used to disable time slicing).

3 Task object FATHER is elaborated, and a task designated %TASK 2 is created. FATHER has no pragma PRIORITY, and thus assumes a default priority. 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 81), in accordance with Ada rules. The elaboration of the task body on lines 27 through 79 defines the statements that tasks of type FATHER_TYPE will execute.

4 Task FATHER declares a single task named CHILD (line 30). A single task represents both a task object and an anonymous task type. Task CHILD is not created or activated until FATHER is activated.

5 The only source of ASTs is this series of TEXT_IO.PUT_LINE statements (input-output completion delivers ASTs).

6 The task FATHER is activated while the main program waits. FATHER has no pragma PRIORITY and this assumes a default priority of 7. (See *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for the rules concerning default priorities.) FATHER's activation consists of the elaboration of lines 27 through 42.

When task FATHER is activated, it waits while its task CHILD is activated and a task designated %TASK 3 is created. CHILD executes one entry call on line 36, and then deadlocks because the entry is never accepted.

Note that because time-slicing is disabled and there are no higher priority runable tasks, FATHER will continue to execute past its activation until it is blocked at the ACCEPT statement at line 45.

7 A single task, MOTHER, is defined, and a task designated %TASK 4 is created. The pragma PRIORITY gives MOTHER a priority of 6.

8 The task MOTHER begins its activation and executes line 89. After MOTHER is activated, the main program (%TASK 1) is eligible to resume its execution. Because %TASK 1 has the default priority 7, which is higher than MOTHER's priority, the main program resumes execution.

9 This is the first rendezvous the main program makes with task FATHER. After the rendezvous FATHER will suspend at the SELECT with TERMINATE statement at line 56.

10 At the third rendezvous with FATHER, FATHER raises the exception SOME_ERROR on line 65. The handler on line 70 catches the exception, aborts the suspended CHILD task, and then reraises the exception; FATHER then terminates.

11 A loop with a delay statement ensures that when control reaches line 120, FATHER has executed far enough to be terminated.

12 This entry call ensures that MOTHER does not wait forever for its rendezvous on line 91. MOTHER executes the accept statement (which involves no other statements), the rendezvous is completed, and MOTHER is immediately switched off the processor at line 92 because its priority is only 6.

13 After its rendezvous with MOTHER, the main program (%TASK 1) executes lines 125 through 127. At line 127, the main program must wait for all its dependent tasks to terminate. When the main program reaches line 127, the only nonterminated task is MOTHER (MOTHER cannot terminate until the **null** statement at line 95 has been executed). MOTHER finally executes to its completion at line 96. Now that all tasks are terminated, the main program completes its execution. The main program then returns and execution resumes with the command-line interpreter.

8.2 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 8–1).

Note

The debugger does not support the task-specific attributes T'CALLABLE, E'COUNT, T'STORAGE_SIZE, and T'TERMINATED. See Section 8.2.4 for more information.

8.2.1 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 8.2.2), 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 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 *OpenVMS Debugger Manual* for more information on path names.

When a task object is elaborated, a task is created by the DEC Ada run-time library, and the task object is assigned its 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 DEC Ada as a procedure. This procedure is called by the DEC 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
```

8.2.2 Task ID (%TASK)

A *task ID* is the value used by the DEC 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$ADARTL+130428
  %TASK 2  7       SUSP  Accept        TASK_EXAMPLE.MOTHER+4
  %TASK 3  7       SUSP  Entry call    TASK_EXAMPLE.FATHER_TYPE$TASK_BODY.CHILD+4
  %TASK 4  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 58 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. Task IDs are never reassigned during the execution of the program.

The DEC Ada run-time library always assigns %TASK 1 to the environment task that executes the main program. On VAX systems, 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. On AXP systems, %TASK 0 is often used for tasks that have been created but are not yet activated.

Note that on VAX systems, task IDs are assigned at task creation; on AXP systems, task ID's are assigned at activation.

8.2.3 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.

8.2.3.1 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 8.5).

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 36 only when line 36 is executed by the task named CHILD:

```
DBG> SET BREAK %LINE 36 WHEN (%ACTIVE_TASK=CHILD)
```

8.2.3.2 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 8.5.

8.2.3.3 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
   .
   .
   .
```

8.2.3.4 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. 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 8–1:

```
DBG> SET BREAK %LINE 59
```

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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for information on AST entry calls.

8.2.4 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 *DEC 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 8.3 for more information on this command.

8.3 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 8.2.

Note that you can use the pragma GET_TASK_INFO to obtain information about the currently executing task. See Section C.2 for more information.

The following sections explain how to use the SHOW TASK command and its qualifiers.

8.3.1 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
  1        2    3     4       5             6
  task id pri hold state   substate     task object
* %TASK 1  7       RUN                SHARE$ADARTL+130428
  %TASK 2  7  HOLD SUSP  Accept       TASK_EXAMPLE.MOTHER+4
  %TASK 3  7       SUSP  Entry call   TASK_EXAMPLE.FATHER_TYPE$TASK_BODY.CHILD+4
  %TASK 4  6       READY              TASK_EXAMPLE.MOTHER
DBG>
```

The information in each column is as follows:

1 The task ID of the task. An asterisk indicates that the task is the visible task.

2 The task priority. DEC Ada priorities range from 0 to 15. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

3 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).

4 Indicates the state of the task when the debugger interrupted program execution. The four possible states recognized by the debugger are identified in Table 8–1. Figure 8–1 shows the possible transitions of a task's state during program execution. Note from the SHOW TASK display that the states of Table 8–1 are abbreviated to RUN, READY, SUSP, and TERM, respectively.

5 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 8–2. 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.

6 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> 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 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 8–1 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 8–1 Task State Transitions

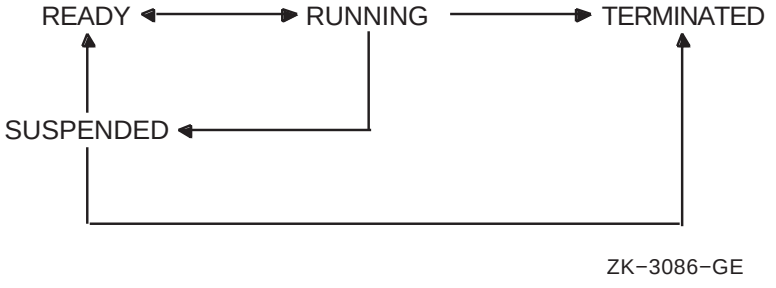


Table 8–2 Task Substates

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.
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 DEC 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]

[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 8–2 (Cont.) Task Substates

Task Substate	Meaning
Terminated	Task terminated normally.
Timed entry call	Task is waiting in a timed entry call.

8.3.2 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.

8.3.2.1 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 8.2.

8.3.2.2 Task-Selection Qualifiers

You can use the task selection qualifiers listed in Table 8–3 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 8–3 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.
/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.

8.3.2.3 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
```

8.3.3 Obtaining Additional Information

You can use the information-selection qualifiers listed in Table 8–4 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 8.3.2.1 through 8.3.2.3.

Table 8–4 SHOW TASK Command Qualifiers for Information Selection

Qualifier	Meaning
/CALLS[=n]	Performs a SHOW CALLS command for each task selected for display (see *OpenVMS Debugger Manual* 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. This qualifier is not fully supported on AXP systems.
/TIME_SLICE	On VAX systems only. 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:

```
1 task id     pri hold state   substate          task object
  %TASK 2      7       SUSP  Accept            TASK_EXAMPLE.FATHER

2       Awaiting rendezvous at: accept START
          having do part at address 00000A68

        Waiting entry callers:
          Waiters for entry BOGUS:
              %TASK 3, type: CHILD
```

```
3       Task type:      FATHER_TYPE
        Created at PC:  TASK_EXAMPLE.%LINE 27
        Parent task:    %TASK 1
        Start PC:       TASK_EXAMPLE.FATHER_TYPE$TASK_BODY
4       Task control block:       5       Stack storage (bytes):
          Task value:   1010344             RESERVED_BYTES:       10640
          Entries:      3                   TOP_GUARD_SIZE:        5120
          Size:         1500                STORAGE_SIZE:         30720
6       Stack addresses:                    Bytes in use:           632
          Top address:  000FB000
          Base address: 001027FC      7 Total storage:          47980
```

The following notes are keyed to this example:

1 Identifying information about the task.

2 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.

3 Task context information.

4 Task control block information. The task value is the address, in decimal notation, of the task control block.

5 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 DEC Ada pragmas TASK_STORAGE and MAIN_STORAGE (VAX only); 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 *DEC Ada Language Reference Manual* and *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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 DEC Ada pragma MAIN_STORAGE (VAX only); 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 *DEC Ada Language Reference Manual* and *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information about this representation

clause and pragma and about the task activation (working) storage area.

- "Bytes in use:" gives the amount of the task stack currently in use.

6 Stack addresses of the task stack.

7 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.

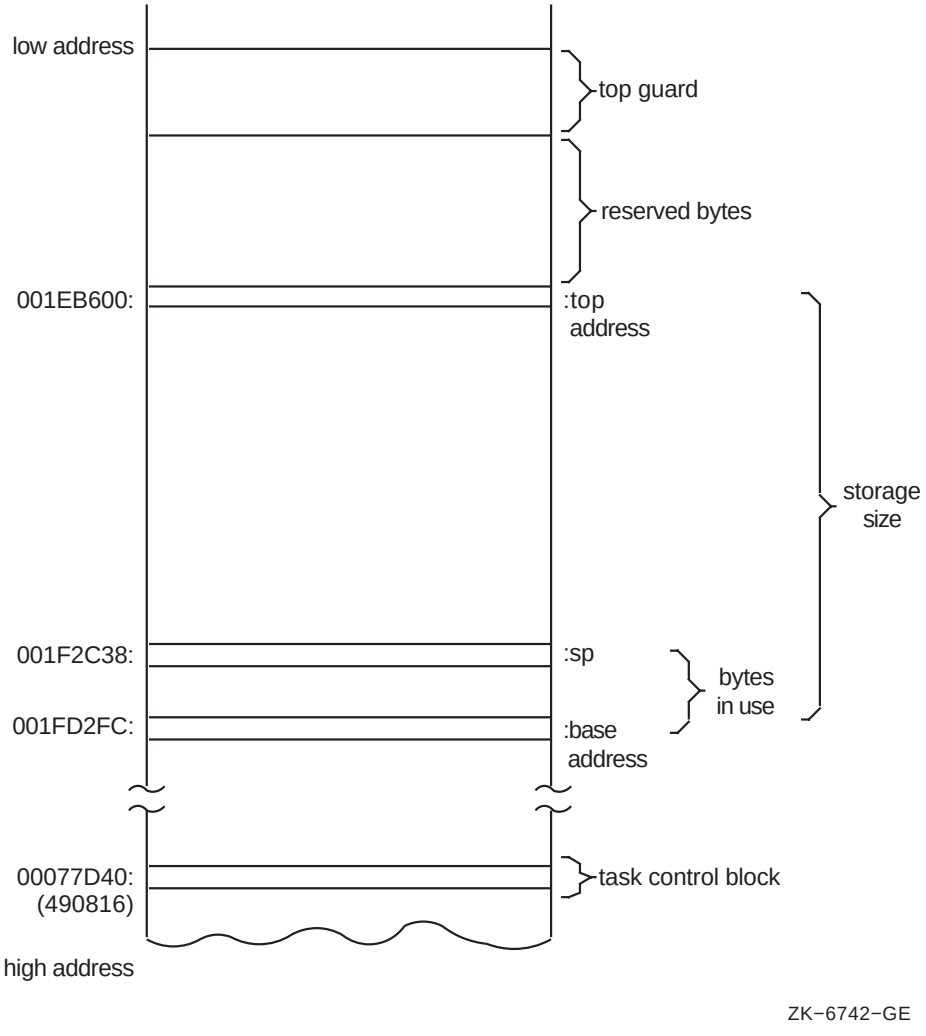
Figure 8–2 shows the task stack for task FATHER.

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.

Figure 8–2 Diagram of a Task Stack



8.4 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 DEC 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>
```

8.5 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 8–5 defines the SET TASK command qualifiers. Section 8.2 defines task expressions. Note that if no qualifier is specified, the /VISIBLE qualifier is assumed by default.

Table 8–5 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 (VAX only) 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 8–5 (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 is used. If no task list is specified, causes the priority of the visible task to be restored.
/TIME_SLICE=t	On VAX systems only. 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 8.7.2). 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 8.2.3.2 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 (on VAX system only) 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
   .
   .
   .
```

8.6 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.

8.6.1 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 53, 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 9 WHEN (%ACTIVE_TASK=FATHER)
```

8.6.2 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 8.2.1, 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 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 the previous 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 previous 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 *OpenVMS Debugger Manual* 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)
```

8.6.3 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 *OpenVMS Debugger Manual* 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 *OpenVMS Debugger Manual* for more information on setting breakpoints and tracepoints; see the *OpenVMS 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 8–6 defines the set of events (event-name keyword values) that apply to DEC 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 8–6 DEC 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 *OpenVMS Debugger Manual*).
HANDLED_OTHERS	Triggers only when an exception is about to be handled in an **others** Ada exception handler (see *OpenVMS Debugger Manual*).
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 DEC Ada run-time library; see the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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 8–6 (Cont.) DEC 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 8–1.)
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 DEC Ada program (or with a program in a supported language that is linked with a DEC 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 *OpenVMS Debugger Manual*). The sample initialization file in Example 8–2 may be useful in helping you to locate task-related errors.

Example 8–2 Sample Debugger Initialization File for DEC 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
!
```

(continued on next page)

Example 8–2 (Cont.) Sample Debugger Initialization File for DEC Ada Tasking Programs

```
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"      !VAX only
!  Define a shorter command for disabling time slicing
!
DEFINE/COMMAND slice="SET TASK/TIME="           !VAX only
!  Define a shorter command for enabling time slicing
```

8.7 Additional Task-Debugging Topics

The following sections discuss additional topics related to task debugging:

- Deadlock
- Time slicing (on VAX systems only)
- Using Ctrl/Y
- Automatic stack checking
- Highlighting task state changes

8.7.1 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 8–7 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 8–7 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

(continued on next page)

Table 8–7 (Cont.) Kinds of Deadlock and Debugger Commands for Diagnosing Them

Kind of Deadlock	Debugger Commands
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

8.7.2 Debugging Programs that Use Time Slicing

Tasking programs that use time slicing 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.

On VAX systems, 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 on VAX systems there is an interaction between DEC 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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems*; for more information on the pragma TIME_SLICE, see the *DEC Ada Language Reference Manual*.

8.7.3 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 DEC 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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for information on this package.

8.7.4 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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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 8.6.1). (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.

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 DCL qualifier conventions (see the *OpenVMS User's 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 DEC 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 DEC Ada compiler to compile one or more DEC 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
/LIBRARY=(lib-term[,...])/PATH	See text.
/PATH	/NOPATH

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]DESIGN[=option]	/[NO]DESIGN
/[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]SMART_RECOMPILATION	/SMART_RECOMPILATION
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/[NO]WARNINGS[=(option[,...])]	See text.

Prompts

_File:

Command Parameters

file-spec[,...]
Specifies one or more DEC 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 DEC 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). DEC Ada source files are compiled in the order in which they appear in the command line. If a source file contains more than one DEC 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 4 for more information on DEC Ada program libraries, sublibraries, and compilation.

Command Qualifiers

/LIBRARY=directory-spec
/LIBRARY=ADA$LIB (D)
Specifies the program library that is to be the current program library for the duration for the compilation. The directory specified must be an existing DEC 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.

/LIBRARY=(lib-term[,..])/PATH

Specifies the library search path that is to be the current path for the duration of the compilation. For more information on library search paths, see Chapter 3.

You can specify *lib-term* as follows:

- The directory specification of a DEC Ada library. For example: [JONES.HOTEL.ADALIB].
- The default path of a DEC Ada library. To specify the default path, you enter the name of a DEC Ada library preceded by an at sign(@). For example: @[JONES.HOTEL.ADALIB].
- A file specification preceded by an at sign (@). For example: @[JONES.HOTEL]MYPATH.TXT. Note the that file, MYPATH.TXT, must contain one or more valid *lib-terms*.

  If you do not specify the full file specification, the default file name is PATH and the default file extension is .TXT.

You must use commas to separate more than one *lib-term* in a library search path. If a term in the value of the /LIBRARY qualifier contains that at sign (@), you must use double quotes (") to surround the term.

/PATH
/NOPATH (D)
Allows you to specify a library search path as the value of the /LIBRARY qualifier.

If you do not specify the /PATH qualifier, the value of the /LIBRARY qualifier must be a directory specification of a DEC Ada library or sublibrary. For more information on library search paths, see Chapter 3.

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 DEC 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 *DEC 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 4). Copied source files may also be used by the debugger (see Chapter 8 for more information on debugging tasks; and the *OpenVMS Debugger Manual* for more information on the debugger).

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 DEC Ada programs with the debugger (see Chapter 8 for more information on debugging Ada tasks; and the *OpenVMS Debugger Manual* for more information on the debugger.

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).

/DESIGN[=option]
/NODESIGN (D)

Allows you to process Ada source files as a detailed program design. For each unit that is design checked without error, the program library is updated with information about that unit. Design-checked units are considered to be obsolete in operations that require full compilation and must be recompiled.

You can request the following options:

[NO]COMMENTS	Determines whether comments are processed for program design information. For the COMMENTS option to have effect, you must specify the /ANALYSIS_DATA qualifier. See *Guide to Source Code Analyzer for VMS Systems* for more information on using the Source Code Analyzer (SCA). If you specify NOCOMMENTS, comments are ignored. On AXP systems, the /DESIGN=COMMENTS qualifier is accepted, but has no effect.
[NO]PLACEHOLDERS	Determines whether design checking is performed. If you specify PLACEHOLDERS, compilation units are design checked—LSE placeholders are allowed and some of the Ada language rules are relaxed so that you can omit some implementation details. If you specify NOPLACEHOLDERS, full compilation is done—the compiler is invoked, LSE placeholders are not allowed, and Ada language rules are not relaxed.

Note that when you specify this option with the /SYNTAX_ONLY qualifier, it determines only whether LSE placeholders are allowed. If you specify NOPLACEHOLDERS, then only valid Ada syntax is allowed.

If you specify the /DESIGN qualifier without supplying any options, the effect is the same as the following default:

```
/DESIGN=(COMMENTS,PLACEHOLDERS)
```

If you specify only one of the options with the /DESIGN qualifier, the default value for the other option is used. For example, /DESIGN=NOCOMMENTS is equivalent to /DESIGN=(NOCOMMENTS,PLACEHOLDERS). In this case, both qualifiers specify that the unit is design-checked, but comment information is not collected. Similarly, /DESIGN=NOPLACEHOLDERS is equivalent to /DESIGN=(COMMENTS,NOPLACEHOLDERS). In this case, both qualifiers specify that comment information is collected, but the unit is not design-checked (that is, in the absence of the /SYNTAX_ONLY qualifier, units are fully compiled).

/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 DEC 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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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. 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, if you specify one of the /OPTIMIZE qualifier primary options on the left (for example, /OPTIMIZE=TIME), it has the same effect as specifying the secondary-option values to the right (in this case, /OPTIMIZE=(TIME,INLINE:NORMAL,SHARE:NORMAL)):

TIME	/OPTIMIZE=(TIME,INLINE:NORMAL,SHARE:NORMAL)
SPACE	/OPTIMIZE=(SPACE,INLINE:NORMAL,SHARE:NORMAL)
DEVELOPMENT	/OPTIMIZE=(DEVELOPMENT,INLINE:NONE, SHARE:NONE)
NONE	/OPTIMIZE=(NONE,INLINE:NONE,SHARE:NONE)

See Chapter 4 for more information on 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 7).

NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the ADA command provides a portability summary (/SHOW=PORTABILITY).

/SMART_RECOMPILATION (D)
/NOSMART_RECOMPILATION
Controls whether smart recompilation information is stored and used to minimize unnecessary recompilations.

When the /SMART_RECOMPILATION qualifier is in effect, detailed information about dependences is stored in the program library for each unit compiled. This information describes the dependences of a unit at a finer level than the compilation unit level.

If smart recompilation is not in effect, detailed information about dependences is not stored in the program library. (See Chapter 5 for more information.)

/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 4 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 previous 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/WARNINGS=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 /WARNINGS=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.

6. $ **ADA/LIBRARY=([JONES.ADALIB],"@[SMITH.ADALIB]")/PATH HOTEL**

   Compiles the compilation units contained in the file HOTEL.ADA using units in [JONES.ADALIB] and in the default path of the [SMITH.ADALIB]. Suppose the default path of [SMITH.ADALIB] identifies the following libraries:

   ```
   [SMITH.ADALIB]
   [PROJECT.ADALIB]
   ```

In this case, the library search path used during the compilation is as follows:

```
[JONES.ADALIB]
[SMITH.ADALIB]
[PROJECT.ADALIB]
```

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 *OpenVMS 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 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
/[NO]OBSOLETE=(option[,...])	/NOOBSOLETE
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/PROCESSING_LEVEL[=option]	See text.
/[NO]SMART_RECOMPILATION	/SMART_RECOMPILATION
/[NO]STATISTICS	/STATISTICS

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 *OpenVMS User's Manual* for more information on wildcard characters.)

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 there are obsolete units in the closure, identifies units that may become obsolete when the obsolete units are recompiled.
6. 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.

/OBSOLETE=(option=[,...])
/NOOBSOLETE (D)
Allows you to ask what the effect on a program or a set of units would be if some specific units were obsolete.

When the execution closure of the units in the parameter list of the command is performed, the units named with the UNIT, SPECIFICATION, and BODY keywords are assumed to be obsolete as described below. If one of those units is not in the execution closure of the units named in the command's parameter list, it is not added to the closure.

Unit names are specified with the UNIT, SPECIFICATION and BODY keywords as follows:

UNIT:(*unit_name*[,...])	The specifications and bodies of units specified with the UNIT keyword are assumed to be obsolete.
SPECIFICATION:(*unit_name*[,...])	Only the specifications of units specified with the SPECIFICATION keyword are assumed to be obsolete.
BODY:(*unit_name*[,...])	Only the bodies of units specified with the BODY keyword are assumed to be obsolete.

You must specify at least one of these keywords with the /OBSOLETE qualifier. Unit names can contain wildcard characters.

By default, units are identified as obsolete based on the current state of the program library.

/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.

/PROCESSING_LEVEL[=option]
Determines the kind of obsolete units identified. Obsolete units are identified based on the level of processing applied to the unit: syntax-checking, design-checking, or full compilation. You can request the following options:

SYNTAX	Determines whether a unit is obsolete because it has been syntax-checked only. Because all units in a program library are at least syntax-checked, and because syntax-checking does not require any particular order of compilation, generally accepts all units as being current.
DESIGN	Determines whether a unit is obsolete because it has been design-checked only. Accepts design-checked units and fully compiled units as being current, unless they are otherwise obsolete (for example, they depend on units that have been syntax-checked only, or they depend on other obsolete units).
FULL	Determines three kinds of obsolete units: units that are obsolete because they have been syntax checked only, units that have been design checked, and units that are obsolete as a result of the compilation of the units they depend on. Units that depend on obsolete units are also considered to be obsolete.

By default, all units are fully checked (/PROCESSING_LEVEL=FULL), and all obsolete units are identified.

/SMART_RECOMPILATION (D)
/NOSMART_RECOMPILATION
Controls whether smart recompilation information, which is stored in the program library, is used to identify obsolete units.

If smart recompilation is not in effect, units are identified as obsolete and in need of recompilation based on their time of compilation only. (See Chapter 5 for more information.)

/STATISTICS (D)
/NOSTATISTICS
Controls whether statistical information is displayed. Statistical information includes the number of obsolete and possibly obsolete units, and the total elasped time for the last compilation of all identified units.

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/OBSOLETE=SPECIFICATION:RESERVATIONS RESERVATIONS
%E, Obsolete library units are detected

%I, The following units need to be recompiled:
RESERVATIONS
    package specification           4-NOV-1992 14:48:39.75 (00:00:03.82)

%I, The following units may also need to be recompiled:
RESERVATIONS
    package body                    4-NOV-1992 14:51:20.11 (00:00:14.02)
RESERVATIONS.RESERVE
    procedure body                  4-NOV-1992 14:49:55.78 (00:00:04.27)
RESERVATIONS.RESERVE.BILL
    procedure body                  4-NOV-1992 14:50:01.55 (00:00:05.12)
RESERVATIONS.CANCEL
    procedure body                  4-NOV-1992 14:51:36.25 (00:00:04.24)

1 obsolete unit, 4 possibly obsolete (total 5)
Total elapsed time for last compilation of all 5 units was 0:00:31.47
```

   This command allows you to ask what the effect would be if you modified the unit RESERVATIONS. In the previous example, the ACS CHECK command lists the units that need to be recompiled, any units that are missing, and the total elasped time for the last compilation of the unit RESERVATIONS.

3. $ **ACS CHECK A, B, C /OBSOLETE=(UNIT:(E, F), BODY:(G, H))**

   Checks the closure of the set of units A, B and C assuming that E and F's specifications and bodies are obsolete, and that G and H's bodies are obsolete.

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]DESIGN[=option]	/NODESIGN
/[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]OBSOLETE=(option[,...])	/NOOBSOLETE
/[NO]OPTIMIZE[=(option[,...])]	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRELOAD	/PRELOAD
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	/QUEUE=ADA$BATCH
/[NO]SHOW[=option]	/SHOW=PORTABILITY
/[NO]SMART_RECOMPILATION	/SMART_RECOMPILATION
/SPECIFICATION_ONLY	See text.
/[NO]STATISTICS	/STATISTICS
/SUBMIT	See text.
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/WAIT	See text.

/[NO]WARNINGS[=(option[,...])]	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 *OpenVMS User's 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.

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 revised external source files to account for new compilation units or unit dependences if the /PRELOAD qualifier (the default) 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. 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 DEC Ada compiler is invoked as follows:

   a. By default (COMPILE/WAIT), the command file is executed in a subprocess. You must wait for the compilation to terminate before entering another command. When this qualifier is in effect, process logical names are propagated to the subprocess generated to execute the command file.

   b. If you specify the /SUBMIT qualifier, the command file generated in step 7 is submitted as a batch job.

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 the terminal by default, or to the a log file if the command file is executed in a batch job (by way of the COMPILE/SUBMIT command).

See Chapter 4 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 command file is executed in batch 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 *OpenVMS User's 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 DEC 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 command file is executed in batch 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 *DEC Ada Language Reference Manual* for more information on the pragmas SUPPRESS and SUPPRESS_ALL.

/CLOSURE

Causes the /SPECIFICATION_ONLY qualifier to apply to all units in the closure of units named in the COMPILE command. (Without the /CLOSURE qualifier, the /SPECIFICATION_ONLY qualifier applies only to the units named in the command.)

See the description of the /SPECIFICATION_ONLY qualifier in the list of command 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.

Note that if you want to get the old behaviour (pre-version 3.0 behavior) for this command, you must also specify the /NOSMART_RECOMPILATION qualifier.

/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 4). Copied source files may also be used by the debugger (see the *OpenVMS Debugger Manual*).

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 DEC Ada programs with the debugger (see Chapter 8 for more information on debugging tasks; see and *OpenVMS Debugger Manual* for more information on the debugger).

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)

/DESIGN[=option]
/NODESIGN (D)

Controls whether a design-level check is performed when identifying obsolete units. A unit is not considered obsolete just because it is design-checked only.

Also directs the compiler to process Ada source files as a detailed program design. For each unit that is design checked without error, the program library is updated with information about that unit. Design-checked units are considered to be obsolete in operations that require full compilation and must be recompiled.

You can request the following options:

[NO]COMMENTS	Determines whether comments are processed for program design information. For the COMMENTS option to have effect, you must specify the /ANALYSIS_DATA qualifier with the ADA command. See *Guide to Source Code Analyzer for VMS Systems* for more information on using the Source Code Analyzer (SCA). If you specify NOCOMMENTS, comments are ignored. On AXP systems, the /DESIGN=COMMENTS qualifier is accepted, but has no effect.

[NO]PLACEHOLDERS	Determines whether design checking is performed. If you specify PLACEHOLDERS, compilation units are design checked—LSE placeholders are allowed and some of the Ada language rules are relaxed so that you can omit some implementation details. If you specify NOPLACEHOLDERS, full compilation is done—the compiler is invoked, LSE placeholders are not allowed, and Ada language rules are not relaxed. Note that when you specify this option with the /SYNTAX_ONLY qualifier, it determines only whether LSE placeholders are allowed. If you specify NOPLACEHOLDERS, then only valid Ada syntax is allowed.

If you specify the /DESIGN qualifier without supplying any options, the effect is the same as the following default:

```
/DESIGN=(COMMENTS,PLACEHOLDERS)
```

If you specify only one of the options with the /DESIGN qualifier, the default value for the other option is used. For example, /DESIGN=NOCOMMENTS is equivalent to /DESIGN=(NOCOMMENTS,PLACEHOLDERS). In this case, both qualifiers specify that the unit is design-checked, but comment information is not collected. Similarly, /DESIGN=NOPLACEHOLDERS is equivalent to /DESIGN=(COMMENTS,NOPLACEHOLDERS). In this case, both qualifiers specify that comment information is collected, but the unit is not design-checked (that is, in the absence of the /SYNTAX_ONLY qualifier, units are fully compiled).

/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 DEC 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 command file is executed in batch 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 DEC 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 command file is executed in batch 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 command file is executed in batch 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.

/OBSOLETE=(option[,...])
/NOOBSOLETE (D)
Affects the overall set of units that is identified as obsolete.

When the execution closure of the units in the parameter list of the command is performed, the units named with the UNIT, SPECIFICATION and BODY keywords are assumed to be obsolete as described below. If one of those units is not in the execution closure of the units named in the command's parameter list, it is not added to the closure.

Unit names are specified with the UNIT, SPECIFICATION, and BODY keywords as follows:

UNIT:(*unit_name*[,...])	The specifications and bodies of units specified with the UNIT keyword are assumed to be obsolete.
SPECIFICATION:(*unit_name*[,...])	Only the specificiations of units specified with the SPECIFICATION keyword are assumed to be obsolete.
BODY:(*unit_name*[,...])	Only the bodies of units specified with the BODY keyword are assumed to be obsolete.

You must specify at least one of these keywords with the /OBSOLETE qualifier. Unit names can contain wildcard characters.

When the /SMART_RECOMPILATION qualifier is in effect, dependent units of the specified units are possibly obsolete and may be recompiled. To force recompilation of dependent units when smart recompilation is in effect, use the /OBSOLETE=UNIT:* qualifier. (See Section 5.1.3 for more information.)

By default, units are identified as obsolete based on the current state of the program library.

/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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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. 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, if you specify one of the /OPTIMIZE qualifier primary options on the left (for example, /OPTIMIZE=TIME), it has the same effect as specifying the secondary-option values to the right (in this case, /OPTIMIZE=(TIME,INLINE:NORMAL,SHARE:NORMAL)):

TIME	/OPTIMIZE=(TIME,INLINE:NORMAL,SHARE:NORMAL)
SPACE	/OPTIMIZE=(SPACE,INLINE:NORMAL,SHARE:NORMAL)
DEVELOPMENT	/OPTIMIZE=(DEVELOPMENT,INLINE:NONE, SHARE:NONE)
NONE	/OPTIMIZE=(NONE,INLINE:NONE,SHARE:NONE)

See Chapter 4 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 (D)
/NOPRELOAD

Controls whether the COMPILE command processes revised 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 processes revised 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 command file is executed in batch 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 command file is executed in batch 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 7).

NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the COMPILE command provides a portability summary (/SHOW=PORTABILITY).

/SMART_RECOMPILATION (D)
/NOSMART_RECOMPILATION
Controls whether smart recompilation information is stored and used to minimize unnecessary recompilations.

When the /SMART_RECOMPILATION qualifier is in effect, detailed information about dependences is stored in the program library for each unit compiled. This information describes the dependences of a unit at a finer level than the compilation unit level.

The ACS COMPILE command uses this information to detect when an unmodified unit in the closure is not affected by changes (if any) in its referenced units that are compiled or recompiled. The ACS COMPILE command does not recompile such dependent units and, thus, minimizes unnecessary recompilations.

Note that the ACS COMPILE command always compiles modified units.

If smart recompilation is not in effect, detailed information about dependences is not stored in the program library, and units are considered obsolete and recompiled based on their time of compilation. (See Chapter 5 for more information.)

/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.

/STATISTICS (D)
/NOSTATISTICS
Controls whether statistical information is displayed. Statistical information includes the number of obsolete and possibly obsolete units, the total elasped time for the last compilation of all identified units, and the estimated elasped time savings due to smart recompilation.

/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 in a subprocess (by way of the COMPILE/WAIT command).

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)
Controls whether a syntax-level check is performed when identifying obsolete units. A unit is not considered obsolete just because it is syntax-checked only. Because all units in a program library are at least syntax-checked, in effect, this qualifier selects only units with revised source files for compilation.

This qualifier also directs the compiler to process source files for syntax only. Other compiler checks are not performed (for example, semantic analysis, type checking, and so on).

By default, the COMPILE command performs full checking when identifying obsolete units and the compiler fully compiles units.)

/WAIT
Directs the program library manager to execute the command file generated 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 executes the command file generated in a subprocess. You must wait for the subprocess to terminate before you can enter another 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 4 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.

Examples

1.
```
ACS> COMPILE/SUBMIT/LOG RESERVATIONS
%I, Invoking the DEC Ada compiler

%I, The following syntax-checked units are obsolete:
RESERVATIONS
    package body                    4-NOV-1992 16:25:34.68 (00:00:14.02)

%I, The following units may also be recompiled:
RESERVATIONS.RESERVE
    procedure body                  4-NOV-1992 14:49:55.78 (00:00:04.27)
RESERVATIONS.RESERVE.BILL
    procedure body                  4-NOV-1992 14:50:01.55 (00:00:05.12)
RESERVATIONS.CANCEL
    procedure body                  4-NOV-1992 14:51:36.25 (00:00:04.24)

1 obsolete unit, 3 possibly obsolete (total 4)
Total elapsed time for last compilation of all 4 units was 0:00:27.65

%I, Job RESERVATIONS (queue CLU_BATCH, entry 388) started on WIDTH_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 MAIN /OBSOLETE=UNIT:*
```

   This combination can be used to force the compilation of the unit MAIN and of all the units in its closure.

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 *OpenVMS User's 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 /SPECIFICATION_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 DEC 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
/FLOAT_REPRESENTATION=option	/FLOAT_REPRESENTATION=VAX_FLOAT
/[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	See text.

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 *OpenVMS User's 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.

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 *OpenVMS User's Manual* and the *VMS Access Control List Editor Manual* for more information on directory protection and ACL options.

Description

The ACS 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:

   FLOAT_REPRESENTATION=VAX_FLOAT
   LONG_FLOAT = G_FLOAT
   MEMORY_SIZE = 2147483647
   SYSTEM_NAME = VAX_VMS or OpenVMS_AXP

   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 DEC 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

/FLOAT_REPRESENTATION=VAX_FLOAT (D)
Initializes the program library to a particular value of FLOAT_ REPRESENTATION. The possible values are either VAX_FLOAT or IEEE_ FLOAT (for AXP systems only). The effect of this qualifier is similar to compiling a pragma FLOAT_REPRESENTATION.

By default, if the /FLOAT_REPRESENTATION qualifier is not specified, the program library is initialized to VAX_FLOAT.

/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 DEC 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 DEC 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 *OpenVMS User's 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 7 for more information on program library protection.

/SYSTEM_NAME=system

Determines the target operating system for the program library. On VAX systems, the possible system values are VAX_VMS and VAXELN. On AXP systems, the system value is OpenVMS_AXP.

By default, if the /SYSTEM_NAME qualifier is not specified, the initial target operating system is VAX_VMS on VAX systems and OpenVMS_AXP on AXP systems.

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 DEC 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
/FLOAT_REPRESENTATION=option	/FLOAT_REPRESENTATION=VAX_FLOAT
/[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	See text.

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 *OpenVMS User's 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.

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 *OpenVMS User's 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 FLOAT_REPRESENTATION, 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

/FLOAT_REPRESENTATION=VAX_FLOAT (D)
Initializes the program library to a particular value of FLOAT_ REPRESENTATION. The possible values are either VAX_FLOAT or IEEE_ FLOAT (for AXP systems only). The effect of this qualifier is similar to compiling a pragma FLOAT_REPRESENTATION.

By default, if the /FLOAT_REPRESENTATION qualifier is not specified, the program library is initialized to VAX_FLOAT.

/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 *OpenVMS User's 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 *OpenVMS DCL Dictionary*).

See Chapter 7 for more information on program library protection.

/SYSTEM_NAME=system

Initializes the target operating system of the program sublibrary. On VAX systems, the possible system values are VAX_VMS and VAXELN. On AXP systems, the system value is OpenVMS_AXP.

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 DEC 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 DEC Ada program library; that is, it must have been created with the ACS CREATE LIBRARY command.

Description

The ACS DELETE LIBRARY command performs the following steps:

1. Checks whether the directory specified to be deleted is a DEC 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 DEC 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 7 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 DEC 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 DEC 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 DEC Ada program sublibrary. If not, a message is issued and there is no further action.
2. If the specified directory is a DEC 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 7 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 *OpenVMS User's Manual* for more information on wildcard characters.)

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.

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 /SPECIFICATION_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 UNIT/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 *OpenVMS User's 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 /SPECIFICATION_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          4-NOV-1992 18:11:47.55
    package body                   4-NOV-1992 18:12:05.46

SCREEN_IO.INPUT
    procedure body                 4-NOV-1992 18:12:20.80

SCREEN_IO.INPUT.BUFFER
    function body                  4-NOV-1992 18:12:56.64

SCREEN_IO.OUTPUT
    procedure body                 4-NOV-1992 18:13:09.14

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                 4-NOV-1992 18:12:20.80
        SCREEN_IO__INPUT.ACU;3
        SCREEN_IO__INPUT.OBJ;3
        SCREEN_IO__INPUT.ADC;3
      @ ADA25$:[RYAN.PROJECT.VAXADA_V30FT3]SCREEN_IO__INPUT.ADA;3
```

```
SCREEN_IO.INPUT.BUFFER
    function body                    4-NOV-1992 18:12:56.64
        SCREEN_IO__BUFFER.ACU;3
        SCREEN_IO__BUFFER.OBJ;3
        SCREEN_IO__BUFFER.ADC;3
      @ ADA25$:[RYAN.PROJECT.VAXADA_V30FT3]SCREEN_IO__BUFFER.ADA;3
```

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.

4.
```
$ ACS DIRECTORY ASSERT,HOTEL,RESERVATIONS.SEND,SQR
ASSERT
    package instantiation          27-OCT-1992 10:07:54.09    <entered>

HOTEL
    procedure body                  4-NOV-1992 16:07:05.22    <main>

RESERVATIONS.SEND
    procedure body                  4-NOV-1992 14:10:23.47    <syntax-checked>

SQR
    function specification          4-NOV-1992 17:10:39.42
    function body                   4-NOV-1992 16:47:02.18    <foreign>

ADA_CALLER
    procedure body                  4-NOV-1992 18:31:40.87    <design-checked>
                                                              <main>

Total of 6 units.
```

Displays the unit name, unit-kind, date-time of last compilation, and unit type and status for each unit specified. Note that the unit ADA_CALLER is a main program that has been design-checked.

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 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 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 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 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 *OpenVMS User's 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 /SPECIFICATION_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	See text.

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 *OpenVMS User's 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 *DEC Ada Language Reference Manual* and *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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 on VAX systems and OpenVMS_AXP on AXP systems. 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 Programming Guide* 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 Programming Guide* 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 *OpenVMS User's 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 /SPECIFICATION_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	See text.
/[NO]TRACEBACK	/TRACEBACK
/[NO]USERLIBRARY[=(table[,...])]	See text.
/WAIT	/WAIT

Parameter Qualifiers	Defaults
/INCLUDE=(object-file,...)	See text.
/LIBRARY	See text.
/OPTIONS	See text.
/SELECTIVE_SEARCH	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 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 *OpenVMS User's 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 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 linker is invoked.
3. Creates a DCL command file for the 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 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 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).

Note that some new qualifiers are available with the linker that are not supported by the ACS LINK command. You can pass such qualifiers to the linker by defining a symbol like the following before invoking the ACS LINK command:

```
$ LINK :== LINK/NATIVE_ONLY/SYSEXE
```

This symbol is then used by the ACS LINK command procedure that invokes the linker.

See Chapter 6 for more information on the ACS LINK command. The linker is described in detail in the *OpenVMS 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 *OpenVMS User's 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 previous 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 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 *OpenVMS 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 DEC 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 DEC Ada units specified.

By default (/MAIN), only one DEC Ada unit may be specified, and that unit must be a DEC 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.

On VAX systems, 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 Programming Guide*.

On AXP systems, the value of NAME is OpenVMS_AXP.

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 *OpenVMS 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 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 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 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.

/SELECTIVE_SEARCH

Omits from the output image symbol table all symbols from the associated input object module that are not needed to resolve outstanding references. The binary code in the object module is always included.

/SHAREABLE

Indicates that the associated input file is a 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 DEC 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 DEC 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 DEC 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 DEC 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 DEC 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.
/[NO]SMART_RECOMPILATION	/SMART_RECOMPILATION
/SUBMIT	See text.
/WAIT	/WAIT

Positional Qualifiers	Defaults
/BACKUP	See text.
/BEFORE[=time]	See text.
/BY_OWNER[=uic]	See text.
/[NO]COPY_SOURCE	/COPY_SOURCE
/CREATED	See text.
/[NO]DESIGN[=option]	/NODESIGN
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[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 DEC 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 *OpenVMS User's Manual* for more information on wildcard characters.)

Description

The ACS LOAD command invokes the DEC 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 4 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 DEC Ada compiler is invoked for syntax-only compilation as follows:
   a. By default (LOAD/WAIT), the command file is executed in a subprocess. You must wait for the compilation to terminate before entering another command. Note that process logical names are propagated to the subprocess generated to execute the command file.
   b. If you specified the /SUBMIT qualifier, the compiler command file generated in step 2 is submitted as a batch job.
   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 *OpenVMS User's 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 /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.

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.

/SMART_RECOMPILATION (D)
/NOSMART_RECOMPILATION
Controls whether smart recompilation information is preserved during the load operation. This information can be used in subsequent operations to minimize recompilations.

When the /SMART_RECOMPILATION qualifier is in effect, smart recompilation information from previous compilations is preserved in the program library for each unit compiled.

If the /NOSMART_RECOMPILATION qualifier is specified, units are loaded without preserving smart recompilation information.

/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 executes the command file generated for the compiler in a subprocess (/WAIT).

/WAIT (D)
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 executes the command file generated for the compiler in a subprocess (LOAD/WAIT).

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 *OpenVMS User's 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 4). Copied source files may also

be used by the debugger (see Chapter 8 for information on debugging tasks; see the *OpenVMS Debugger Manual* for information on debugger).

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.

/DESIGN[=option]
/NODESIGN (D)
Allows you to process Ada source files as a detailed program design.

You can request the following options:

[NO]COMMENTS	Determines whether comments are processed for program design information. This option can be specified with the ACS LOAD command, however, it does not have an effect. On AXP systems, the /DESIGN=COMMENTS qualifier is accepted, but has no effect.
[NO]PLACEHOLDERS	Determines whether LSE placeholders are allowed. If you specify NOPLACEHOLDERS, then only valid Ada syntax is allowed.

If you specify the /DESIGN qualifier without supplying any options, the effect is the same as the following default:

```
/DESIGN=(COMMENTS,PLACEHOLDERS)
```

If you specify only one of the options with the /DESIGN qualifier, the default value for the other option is used. For example, /DESIGN=NOCOMMENTS is equivalent to /DESIGN=(NOCOMMENTS,PLACEHOLDERS). In this case, both qualifiers specify that placeholders are allowed. Similarly, /DESIGN=NOPLACEHOLDERS is equivalent to /DESIGN=(COMMENTS, NOPLACEHOLDERS). In this case, both qualifiers have no effect.

/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 DEC 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 DEC 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 *OpenVMS User's 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 4 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 previous 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 library, that are to be merged into the next library in the current path. 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 *OpenVMS User's 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 that library's index file.

If the parent library already has a version of the unit to be merged, the unit to be merged must have a more recent external source file.

If you specified the ACS SET LIBRARY/PATH command to set the current path, the ACS MERGE command moves one or more units from the current program library to the next library in the current path. In the case of sublibraries, the next library in the current path is the parent library unless you explicitly set the path otherwise. For more information on library search paths, see Chapter 3.

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 current program library 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 /SPECIFICATION_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.

MODIFY LIBRARY

Modifies the default path of a DEC Ada program library or program sublibrary.

Format

MODIFY LIBRARY/PATH lib-term[,...]

Command Qualifiers	Defaults
/[NO]EDIT[=edit-cmd]	/NOEDIT
/LIBRARY=directory-spec	See text.
/[NO]PATH	/NOPATH
/[NO]VERIFY	/NOVERIFY

Command Parameters

lib-term[,...]
Specifies the new default library search path for the specified library. For more information on library search paths, see Chapter 3.

You can specify *lib-term* as follows:

- The directory specification of a DEC Ada library. For example: [JONES.HOTEL.ADALIB].
- The default path of a DEC Ada library. To specify the default path, you enter the name of a DEC Ada library preceded by an at sign(@). For example: @[JONES.HOTEL.ADALIB].
- A file specification preceded by an at sign (@). For example: @[JONES.HOTEL]MYPATH.TXT. Note the that file, MYPATH.TXT, must contain valid library terms.

  If you do not specify a full file specification, the default file name is PATH and the default file extension is .TXT.

You must use commas to separate more than one *lib-term* in a library search path.

Description

The ACS MODIFY LIBRARY/PATH command redefines the default library search path for a given library. If you do not specify a library (you omit the /LIBRARY=*directory-spec* qualifier), the default path for the current program library is redefined.

The ACS MODIFY LIBRARY/PATH command stores the specified library search path in the given library in its original form. In other words, the search path is stored exactly as you specified it on the command line or with the editor. The command also evaluates and verifies the new library search path, and reports any errors.

If you enter the ACS MODIFY LIBRARY/PATH command interactively (that is, at the ACS> prompt), the current path is not reevaluated. Thus, if you modified the default path for the current library, this modification will not take effect until you reinvoke ACS.

For more information on library search paths, see Chapter 3.

Command Qualifiers

/EDIT[=edit_cmd]
/NOEDIT(D)
Invokes an editor which allows you to edit the default path associated with the current program library. The current definition of the default path for the given library is placed in a text file, and the editor specified by *edit-cmd* is invoked. If you do not specify an editor with the /EDIT qualifier, callable EDT is invoked. When you exit from the editor, the default path for the library is redefined to be the library search path contained in the edited file.

/LIBRARY=directory-spec
Specifies a library whose default library search path is to be modified. By default, the default path of the current program library is modified.

/PATH
/NOPATH (D)
Specifies that the default library search path for the given library is to be modified.

/VERIFY(D)
/NOVERIFY
Controls whether the program library manager suppresses the evaluation and verification of the new default path.

By default, the new default path is evaluated and verified.

Example

```
ACS> MODIFY LIBRARY/PATH/LIBRARY=[MY_LIB] [MY_LIB],[LIB1],[LIB2]
```

Redefines the default path of the library [MY_LIB] to be as follows:

```
[MY_LIB]
[LIB1]
[LIB2]
```

Note that if you enter the ACS MODIFY LIBRARY command interactively, the current path is not reevaluated. Thus, if you modified the default path for the current library, this modification will not take effect until you reinvoke ACS.

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]DESIGN[=option]	/NODESIGN
/[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]OBSOLETE=(option[,...])	/NOOBSOLETE
/[NO]OPTIMIZE[=(option[,...])]	See text.
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	/QUEUE=ADA$BATCH
/[NO]SHOW[=option]	/SHOW=PORTABILITY
/[NO]SMART_RECOMPILATION	/SMART_RECOMPILATION

/SPECIFICATION_ONLY	See text.
/[NO]STATISTICS	/STATISTICS
/SUBMIT	See text.
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/WAIT	/WAIT
/[NO]WARNINGS[=(option[,...])]	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 *OpenVMS User's 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.

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 /OBSOLETE qualifier, can be used to force the recompilation of the entire execution closure of a set of units.

The RECOMPILE 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. If all units in the closure are in the program library and are current, issues an informational message and terminates the operation.
4. Identifies any unit in the closure that is missing from the current program library.
5. Identifies any unit in the closure that is obsolete and must be recompiled.
6. If any units in the closure 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.
7. Unless the /COMMAND qualifier is specified, invokes the DEC Ada compiler as follows:

    a. By default (RECOMPILE/WAIT), the command file is executed in a subprocess. You must wait for the compilation to terminate before entering another command. When the /WAIT qualifier is in effect, process logical names are propagated to the subprocess generated to execute the command file.

    b. If you specify the /SUBMIT qualifier, the compiler command file generated in step 2 is submitted as a batch job.

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 to your terminal, by default, or to a log file if the command file is executed in batch mode (by way of the RECOMPILE/SUBMIT command).

See Chapter 4 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 command file is executed in batch 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 *OpenVMS User's 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 DEC 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 command file is executed in batch 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 *DEC Ada Language Reference Manual* for more information on the pragmas SUPPRESS and SUPPRESS_ALL.

/CLOSURE
Causes the /SPECIFICATION_ONLY to apply to all units in the closure of units named in the RECOMPILE command. (Without the /CLOSURE qualifier, the /SPECIFICATION_ONLY qualifier applies only to the units named in the command.)

See the description of the /SPECIFICATION_ONLY qualifier in the list of command 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 4). Copied source files may also be used by the debugger (see Chapter 8 for more information on debugging tasks; see *OpenVMS Debugger Manual* for more information on the debugger).

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 DEC Ada programs with the debugger (see Chapter 8 for more information on debugging tasks; see the *OpenVMS Debugger Manual* for more information on the debugger).

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).

/DESIGN[=option]
/NODESIGN (D)

Controls whether a design-level check is performed when identifying obsolete units. A unit is not considered obsolete just because it is design-checked only.

Also directs the compiler to process Ada source files as a detailed program design. For each unit that is design checked without error, the program library is updated with information about that unit. Design-checked units are considered to be obsolete in operations that require full compilation and must be recompiled.

You can request the following options:

[NO]COMMENTS	Determines whether comments are processed for program design information. For the COMMENTS option to have effect, you must specify the /ANALYSIS_DATA qualifier with the ADA command. See *Guide to Source Code Analyzer for VMS Systems* for more information on using the Source Code Analyzer (SCA). If you specify NOCOMMENTS, comments are ignored. On AXP systems, the /DESIGN=COMMENTS qualifier is accepted, but has no effect.
[NO]PLACEHOLDERS	Determines whether design checking is performed. If you specify PLACEHOLDERS, compilation units are design checked—LSE placeholders are allowed and some of the Ada language rules are relaxed so that you can omit some implementation details. If you specify NOPLACEHOLDERS, full compilation is done—the compiler is invoked, LSE placeholders are not allowed, and Ada language rules are not relaxed. Note that when you specify this option with the /SYNTAX_ONLY qualifier, it determines only whether LSE placeholders are allowed. If you specify NOPLACEHOLDERS, then only valid Ada syntax is allowed.

If you specify the /DESIGN qualifier without supplying any options, the effect is the same as the following default:

```
/DESIGN=(COMMENTS,PLACEHOLDERS)
```

If you specify only one of the options with the /DESIGN qualifier, the default value for the other option is used. For example, /DESIGN=NOCOMMENTS is equivalent to /DESIGN=(NOCOMMENTS,PLACEHOLDERS). In this case, both qualifiers specify that the unit is design-checked, but comment information is not collected. Similarly, /DESIGN=NOPLACEHOLDERS is equivalent to /DESIGN=(COMMENTS,NOPLACEHOLDERS). In this case, both qualifiers specify that comment information is collected, but the unit is not design-checked (that is, in the absence of the /SYNTAX_ONLY qualifier, units are fully compiled).

/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 DEC 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 command file is executed in batch 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 DEC 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 command file is executed in batch 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 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.

/OBSOLETE=(option[,...])
/NOOBSOLETE (D)
Affects the overall set of units that is identified as obsolete.

When the execution closure of the units in the parameter list of the command is performed, the units named with the UNIT, SPECIFICATION, and BODY keywords are assumed to be obsolete as described below. If one of those units is not in the execution closure of the units named in the command's parameter list, it is not added to the closure.

Unit names are specified with the UNIT, SPECIFICATION, and BODY keywords as follows:

UNIT:(*unit_name*[,...])	The specifications and bodies of units specified with the UNIT keyword are assumed to be obsolete.
SPECIFICATION:(*unit_name*[,...])	Only the specifications of units specified with the SPECIFICATION keyword are assumed to be obsolete.
BODY:(*unit_name*[,...])	Only the bodies of units specified with the BODY keyword are assumed to be obsolete.

You must specify at least one of these keywords. Unit names can contain wildcard characters.

When the /SMART_RECOMPILATION qualifier is in effect, dependent units of the specified unit may or may not be recompiled. To force recompilation of dependent units when smart recompilation is in effect, use the /OBSOLETE=UNIT:* qualifier. (See Section 5.1.3 for more information.)

By default, units are identified as obsolete based on the current state of the program library.

/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 *DEC Ada Run-Time Reference Manual for OpenVMS Systems* 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. 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, if you specify one of the /OPTIMIZE qualifier primary options on the left (for example, /OPTIMIZE=TIME), it has the same effect as specifying the secondary-option values to the right (in this case, /OPTIMIZE=(TIME,INLINE:NORMAL,SHARE:NORMAL)):

TIME	/OPTIMIZE=(TIME,INLINE:NORMAL,SHARE:NORMAL)
SPACE	/OPTIMIZE=(SPACE,INLINE:NORMAL,SHARE:NORMAL)
DEVELOPMENT	/OPTIMIZE=(DEVELOPMENT,INLINE:NONE, SHARE:NONE)
NONE	/OPTIMIZE=(NONE,INLINE:NONE,SHARE:NONE)

See Chapter 4 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.

/PRINTER[=queue-name]
/NOPRINTER (D)

Controls whether the batch job log file is queued for printing when the command file is executed in batch 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 command file is executed in batch 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 7).
NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the RECOMPILE command provides a portability summary (/SHOW=PORTABILITY).

/SMART_RECOMPILATION (D)
/NOSMART_RECOMPILATION
Controls whether smart recompilation information is stored and used to identify obsolete units.

When the /SMART_RECOMPILATION qualifier is in effect, detailed information about dependences is stored in the program library for each unit compiled. This information describes the dependences of a unit at a finer level than the compilation unit level.

The ACS RECOMPILE command uses this information to detect when a unit in the closure is not affected by changes (if any) in its referenced units that are recompiled. The ACS RECOMPILE command does not recompile such dependent units and thus minimizes unnecessary recompilations.

If smart recompilation is not in effect, detailed information about dependences is not stored in the program library, and units are considered obsolete and recompiled based on their time of compilation. (See Chapter 5 for more information.)

/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.

/STATISTICS (D)
/NOSTATISTICS
Controls whether statistical information is displayed during recompilation. Statistical information includes the number of obsolete and possibly obsolete units, the total elasped time for the last compilation of the identified units, and the estimated elasped time savings due to smart recompilation.

/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 executes the command file for the compiler in a subprocess (by way of the RECOMPILE/WAIT command).

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)
Controls whether a syntax-level check is performed when identifying obsolete units. A unit is not considered obsolete just because it is syntax-checked only. Because all units in a program library are at least syntax-checked, in effect, this qualifier generally identifies all units as current.

The /SYNTAX_ONLY qualifier also directs the compiler to process source files for syntax only. Other compiler checks are not performed (for example, semantic analysis, type checking, and so on).

By default, the RECOMPILE command performs full checking when identifying obsolete units (and the compiler fully compiles units).

/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 executes the command file generated for the compiler to a subprocess: you must wait for the subprocess to terminate before you can enter another 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 4 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 previous 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.

Examples

1.
```
ACS> RECOMPILE/SUBMIT/LOG HOTEL
%I, The following syntax-checked units are obsolete:
RESERVATIONS
    package specification           4-NOV-1992 20:00:45.97
HOTEL
    procedure body                  4-NOV-1992 20:05:16.26

2 obsolete units

%I, Job HOTEL (queue ALL_BATCH, entry 448) started on FAST_BATCH
```

   Lists all of the units in the closure of unit HOTEL that need to be recompiled, then submits the compiler command file generated by the program library manager as a batch job.

2.
```
ACS> RECOMPILE/CLOSURE/NOCHECK/COMMAND HOTEL
```

   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.

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 *OpenVMS User's Manual* for more information on wildcard characters.)

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 /SPECIFICATION_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 DEC 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 DEC 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.

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 DEC Ada program library or program sublibrary as the current program library.

Format

SET LIBRARY directory-spec (Default)

SET LIBRARY/PATH lib-term[,...]

Command Qualifiers	Defaults
/[NO]EXCLUSIVE	/NOEXCLUSIVE
/[NO]LOG	/LOG
/[NO]READ_ONLY	/NOREAD_ONLY
/[NO]VERIFY	/VERIFY

Prompts

_Library:
_Path:

Command Parameters

directory-spec
You use this command parameter when you do not also specify the /PATH qualifier.

This parameter specifies the program library or program sublibrary that defines the current program library. The directory you specify must be a valid DEC Ada program library or program sublibrary, previously created with the CREATE LIBRARY or CREATE SUBLIBRARY command, respectively.

lib-term
You use this command parameter when you specify the /PATH qualifier.

This parameter specifies one or more valid library terms (*lib-term*) that are to be defined as the current path. You can specify *lib-term* as follows:

- The directory specification of DEC Ada library or sublibrary.
- The default path of a DEC Ada library. To specify the default path, you enter the name of a DEC Ada library preceded by an at sign(@).
- A file specification preceded by an at sign (@).

For more information on library search paths, Chapter 3.

Description

The ACS SET LIBRARY command defines the current program library. DEC 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 DEC 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. The program library manager and the compiler use that logical name to maintain the current program library context when performing various operations.

   This assignment takes place even if the specified directory is invalid. 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).

You use the second form (ACS SET LIBRARY/PATH) to explicitly specify the current path. When you use this form, the first library in the specified path is defined as the current program library. For more information on libraries and library search paths, see Chapters 2 and 3.

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.

/PATH
/NOPATH (D)
Allows you to define a current path that differs from the default path associated with the current program library.

You use this qualifier to specify the second form of the ACS SET LIBRARY command. See the description section of this command for more information.

/LOG (D)
/NOLOG
Controls whether the directory specification of the current program library being defined is displayed.

By default, the 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.

/VERIFY (D)
/NOVERIFY
Controls whether the current path is evaluated and verified when an ACS SET LIBRARY command is entered.

By default, the current path is evaluated and verified each time you enter an ACS SET LIBRARY command.

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/PATH [JONES.HOTEL.ADALIB],@[SMITH.ADALIB]
```

   Defines the program library [JONES.HOTEL.ADALIB] and establishes the current path. Suppose the default path of [SMITH.ADALIB] is as follows:

```
[SMITH.ADALIB]
[PROJECT.ADALIB]
```

   In this case, the current path evaluates to the following:

```
[JONES.ADALIB]
[SMITH.ADALIB]
[PROJECT.ADALIB]
```

3.
```
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 FLOAT_REPRESENTATION, 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
/FLOAT_REPRESENTATION=option	/FLOAT_REPRESENTATION=VAX_FLOAT
/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:

- FLOAT_REPRESENTATION=VAX_FLOAT
- LONG_FLOAT = G_FLOAT
- MEMORY_SIZE = 2147483647
- SYSTEM_NAME = VAX_VMS or OpenVMS_AXP

These may be changed by compiling a unit that contains the pragmas FLOAT_REPRESENTATION, 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

/FLOAT_REPRESENTATION=VAX_FLOAT (D)

Redefines the value of the program library characteristic FLOAT_REPRESENTATION. The possible values are either VAX_FLOAT or IEEE_FLOAT (for AXP systems only).

By default, the current value of FLOAT_REPRESENTATION is unchanged.

/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 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 DEC 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 DEC 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, unevaluated and evaluated forms of the current and default paths, and the values of the program library characteristics FLOAT_REPRESENTATION, LONG_FLOAT, MEMORY_SIZE, and SYSTEM_NAME. For more information on paths, see Chapter 3.

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
Current program library DISK:[LIB2]

Current path in its original form:

    DISK:[LIB2]
    @DISK:[LIB1]

Current path evaluates to:

    DISK:[LIB2]
    DISK:[LIB1]
    DISK:[LIB0]

Program library DISK:[LIB2]

Created:           4-NOV-1992 16:33:30.74, by DEC Ada V3.0
Last reorganized:  4-NOV-1992 19:47:36.13

Default path in its original form:
    DISK:[LIB2]
    @DISK:[LIB1]

which evaluates to:

    DISK:[LIB2]
    DISK:[LIB1]
    DISK:[LIB0]

Pragmas that affect STANDARD and SYSTEM:

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

Identifies DISK:[LIB2] as the current program library with a current path of DISK:[LIB2],@DISK:[LIB1]. This path evaluates to DISK:[LIB2], DISK:[LIB1], DISK:[LIB0]. This example also shows the pragmas that affect STANDARD and SYSTEM.

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]OBSOLETE=(option[,...])	/NOOBSOLETE
/[NO]PORTABILITY	/NOPORTABILITY
/PROCESSING_LEVEL[=option]	See text.
/[NO]SMART_RECOMPILATION	/SMART_RECOMPILATION

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 *OpenVMS User's 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 FLOAT_ REPRESENTATION, 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.

/OBSOLETE=(option[,...])
/NOOBSOLETE (D)
Allows you to ask what the effect on a program or a set of units would be if some specific units were obsolete.

When the execution closure of the units in the parameter list of the command is performed, the units named with the UNIT,SPECIFICATION, and BODY keywords are assumed to be obsolete as described below. If one of those units is not in the execution closure of the units named in the command's parameter list, it is not added to the closure.

Unit names are specified with the UNIT, SPECIFICATION, and BODY keywords as follows:

UNIT:(*unit_name*[,...])	The specifications and bodies of units specified with the UNIT keyword are assumed to be obsolete.
SPECIFICATION:(*unit_name*[,...])	Only the specifications specified by the SPECIFICATION keyword are assumed to be obsolete.
BODY:(*unit_name*[,...])	Only the bodies of units specified with the BODY keyword are assumed to be obsolete.

You must specify at least one of these keywords. Unit names can contain wildcard characters.

By default, units are identified as obsolete based on the current state of the program library.

/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 4 for a discussion of portability.

/PROCESSING_LEVEL[=option]
Determines the kind of obsolete units identified. Obsolete units are identified based on the level of processing applied to the unit: syntax checking, design checking, or full compilation. You can request the following options:

SYNTAX	Determines whether a unit is obsolete because it has been syntax-checked only. Because all units in a program library are at least syntax-checked, and because syntax-checking does not require any particular order of compilation, generally accepts all units as being current.
DESIGN	Determines whether a unit is obsolete because it has been design-checked only. Accepts design-checked units and fully compiled units as being current, unless they are otherwise obsolete (for example, they depend on units that have been syntax-checked only, or they depend on other obsolete units).
FULL	Determines three kinds of obsolete units: units that are obsolete because they have been syntax-checked only, units that have been design-checked, and units that are obsolete as a result of the compilation of the units they depend on. Units that depend on obsolete units are also considered to be obsolete.

By default, all units are fully checked (/PROCESSING_LEVEL=FULL), and all obsolete units are identified.

/SMART_RECOMPILATION (D)
/NOSMART_RECOMPILATION
Controls whether smart recompilation information, which is stored in the program library, is used to identify obsolete units.

If smart recompilation is not in effect, units are identified as obsolete and in need of recompilation based on their time of compilation only. (See Chapter 5 for more information.)

Example

```
ACS> SHOW PROGRAM/PORTABILITY ADA_CALLER

ADA_CALLER
ADA_CALLER
4-NOV-1992 08:57:12.48

Program library USER:[TEST]

Created:           1-NOV-1992 10:03:53.93, by DEC Ada V3.0
Last reorganized: <No reorganization date>

Default path in its original form:

    USER:[TEST]

Default path evaluates to:

    USER:[TEST]

Pragmas that affect STANDARD and SYSTEM:

    pragma FLOAT_REPRESENTATION(VAX_FLOAT)
    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: 4-NOV-1992 08:56:42.94
 Source file: 31-JUL-1992 16:23:43.39  USER:[TEST]ADA_CALLER.ADA;1
 With list: SQR
   INTEGER_TEXT_IO

INTEGER_TEXT_IO
    Package instantiation
 Compiled:  2-NOV-1992 01:47:11.30
 Entered from: ADA$PREDEFINED_ROOT:[ADALIB]
 With list: TEXT_IO
 Inline_Generic: TEXT_IO

IO_EXCEPTIONS
    Package specification
 Compiled:  2-NOV-1992 01:45:42.02
 Entered from: ADA$PREDEFINED_ROOT:[ADALIB]

SQR
    Function specification
 Compiled: 4-NOV-1992 08:51:26.95
 Source file:  7-NOV-1988 17:06:41.30  USER:[TEST]SQR_.ADA;2
    Foreign function body
        Object file:    4-NOV-1992 08:51:26.95  SQR.OBJ;1
```

```
SYSTEM
    Builtin package

TEXT_IO
    Package specification
 Compiled:  2-NOV-1992 01:46:43.02
 Entered from: ADA$PREDEFINED_ROOT:[ADALIB]
 With list: IO_EXCEPTIONS

    Package body
 Compiled:  2-NOV-1992 01:46:56.16
 Entered from: ADA$PREDEFINED_ROOT:[ADALIB]
 With list: SYSTEM


PORTABILITY SUMMARY

predefined SHORT_INTEGER or SHORT_SHORT_INTEGER or SHORT_SHORT_SHORT_INTEGER
   SYSTEM spec

with SYSTEM             TEXT_IO body

predefined floating types in package SYSTEM*
   TEXT_IO body

enumeration representation clause
   SYSTEM spec             TEXT_IO spec

length SIZE representation clause
   SYSTEM spec

record representation clause
   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

pragma INLINE_GENERIC*  TEXT_IO spec

pragma PACK             SYSTEM spec

 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 DEC 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 DEC 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 DEC Ada (compiler and program library manager) that is installed on your system.

Example

```
ACS> SHOW VERSION
DEC Ada V3.0-0
```

Shows that Version 3.0 of DEC 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 *OpenVMS 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 DEC 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 7).

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.

B

Comparison of DEC Ada Commands for ULTRIX and VMS Systems

DEC Ada provides a comparable set of compilation and program library manager commands for ULTRIX and VMS systems. The behavior of these commands is similar, however, the spelling and action of each individual command is operating system specific. Table B–1 lists the DEC Ada commands on ULTRIX and VMS systems.

For more information on DEC Ada commands for ULTRIX systems, see *Developing Ada Programs on ULTRIX Systems.*

Table B–1 Comparison of DEC Ada Commands on ULTRIX and VMS Systems

ULTRIX Systems	VMS Systems
`acat`	ACS EXTRACT SOURCE
`acp`	ACS COPY UNIT
`acp -d`	ACS MERGE
`ada`	ADA[1]
`ada -y`	ACS LOAD
`aimport`	ACS COPY FOREIGN ACS ENTER FOREIGN
`ald`	ACS LINK
`ald -r`	ACS EXPORT
`als`	ACS DIRECTORY
`amake`	ACS COMPILE
`amake -L`	ACS LINK

[1]This is a DCL and not DEC Ada command.

(continued on next page)

Table B–1 (Cont.) Comparison of DEC Ada Commands on ULTRIX and VMS Systems

ULTRIX Systems	VMS Systems
`amake -nu`	ACS CHECK
`amklib`	ACS CREATE LIBRARY
`amklib -p`	ACS CREATE SUBLIBRARY
`aprintlib`	ACS SHOW LIBRARY
`areport`	ACS SHOW PROGRAM
`arm`	ACS DELETE UNIT
`armlib`	ACS DELETE LIBRARY, ACS DELETE SUBLIBRARY
`axargs`	N/A
`man`[2] with appropriate DEC Ada command	ACS HELP
`printenv ADASRC`[3]	ACS SHOW SOURCE
`setenv ADALIB`[3]	ACS SET LIBRARY
`setenv ADASRC`[3]	ACS SET SOURCE
`setenv ADASRC`[3]`","; amake`	ACS RECOMPILE
N/A	ACS ATTACH
N/A[4]	ACS ENTER UNIT
N/A	ACS EXIT
N/A[4]	ACS REENTER
N/A	ACS REORGANIZE
N/A[5]	ACS SET PRAGMA
N/A	ACS SHOW VERSION
N/A	ACS SPAWN

[2]This is an ULTRIX command, not a DEC Ada command.

[3]On ULTRIX systems, to establish a program library context, you define the ADALIB environment variable to be the desired program library context. Alternatively, you can use the `-A` *`context`* option with the appropriate DEC Ada command. Note that the `setenv` command is a C shell and not a DEC Ada command.

[4]On VMS systems, several features (such as the ACS ENTER UNIT and REENTER commands, sublibraries, and library search paths) are provided which help you work with multiple program libraries. On ULTRIX systems, you use program library contexts when working with multiple program libraries. Program library contexts are very similar to library search paths. For more information on library search paths, see Chapter 3 of this manual.

[5]To get the same effect as the ACS SET PRAGMA command, you must compile a source file containing the desired pragma.

(continued on next page)

Table B–1 (Cont.) Comparison of DEC Ada Commands on ULTRIX and VMS Systems

ULTRIX Systems	VMS Systems
N/A	ACS VERIFY

C

Supplemental Information for Debugging Ada Programs

This appendix provides a sample debugging session and describes the DEC Ada predefined package GET_TASK_INFO, which can be used when debugging DEC Ada tasks. For more information on using the debugger, see the *OpenVMS Debugger Manual*.

C.1 Sample Debugging Session

The following shows a sample debugging session with a DEC 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

             OpenVMS DEBUG Version V5.5

%I, language is ADA, module set to ADD_INTEGERS
%I, 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
%I, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EXIT   12
$
```

The actions in the previous 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.

2 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).

3 The first GO command executes the program's elaboration code, and breaks at the main program; the next GO command starts program execution.

4 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.

5 Use the EXAMINE command to determine the current value of the variable TOTAL. Its value is 0, as expected at this point.

6 The GO command resumes program execution. The program now prompts you for a number. You type 5. The program's response is correct.

7 The debugger again reaches the breakpoint at the first executable line inside the loop and displays the source line.

8 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.

9 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.

10 The GO command resumes program execution. The result is correct.

11 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.

12 The EXIT command terminates the debugging session.

You can now correct the program so that it reinitializes the variable TOTAL correctly.

C.2 Using the Package GET_TASK_INFO

The package GET_TASK_INFO provides an interface that allows you to obtain information about the currently executing task. This package may provide a useful way of obtaining information in situations where using the debugger is impractical.

Table C–1 shows the functions that you can call to the package GET_TASK_INFO.

Table C–1 GET_TASK_INFO Functions

Function	Description
GET_CURRENT_TASK_ID	Returns the current task's TASK ID.
GET_CURRENT_TASK_PARENT_ID	Returns the TASK ID of the parent of current task. If the task has no parent, then zero is returned.
GET_CURRENT_TASK_CREATED_AT_PC	Returns the PC where the current task was created.
GET_CURRENT_TASK_STACK_TOP	Returns the top of the current task's stack. When this function is called from a main task with an extendable stack, this function returns zero.

(continued on next page)

Table C–1 (Cont.) GET_TASK_INFO Functions

Function	Description
GET_CURRENT_TASK_STACK_BASE	Returns the base of the current task's stack. When this function is called from a main task with an extendable stack, this function returns zero.
GET_CURRENT_TASK_TYPE_NAME	Returns the current task's type name.

For more information, see the package specification for GET_TASK_INFO.

D

Program Design Language Support

DEC Ada includes support for processing Ada source code as a detailed program design. This capability allows you to use DEC Ada as a Program Design Language (PDL) processor, when used with the Language-Sensitive Editor (LSE) and Source Code Analyzer (SCA).

See the *Guide to Language-Sensitive Editor for VMS Systems* for more information on working with LSE; see *Guide to Source Code Analyzer for VMS Systems* for more information working with SCA.

D.1 Program Design Support

DEC Ada provides several qualifiers that allow you to use DEC Ada as a program design processor. To process Ada source code as a detailed program design, use the /DESIGN qualifier with the DCL ADA, ACS COMPILE, LOAD, and RECOMPILE commands. An additional qualifier, /PROCESSING_LEVEL, can be used with the ACS CHECK or ACS SHOW PROGRAM command to determine the kind of obsolete units identified. The /ANALYSIS_DATA and /DESIGN qualifiers with the DCL ADA, ACS COMPILE and RECOMPILE commands allow you to include design information in comments in the SCA analysis data file.

Used with LSE and SCA, these qualifiers provide an integrated software development environment that includes the low-level design phase of the software development life cycle. In other words, you can use DEC Ada as your Program Design Language (PDL) processor.

In the VMS environment, you create detailed designs as follows:

- Use DEC Ada or another DEC programming language
- Embed design information in comments
- Write algorithms with pseudocode and regular placeholders

Note that you cannot link or run detailed designs.

With LSE, you can use pseudocode placeholders to express design information. DEC Ada accepts the special 8-bit double angle brackets, « and », or 7-bit combined angle bracket and vertical bar characters, <| and |>, to delimit pseudocode placeholders.

Note

The 8-bit characters may appear as other characters (for example, + or ;) with output devices that do not support them.

The 8-bit characters are the default characters used and recognized by the LSE. To use the 7-bit pseudocode placeholder delimiters, you must define them in a LSE environment file.

You can express other design information in tagged comments. In addition, you can convert pseudocode placeholders into comments and store the design information in SCA libraries.

If you specify the /DESIGN=PLACEHOLDERS qualifier with the DCL ADA command, your program is design checked. Design checking means that your program is checked for internal consistency, and pseudocode and regular placeholders are accepted (in well-defined contexts) as valid program syntax. Design checking differs from full compilation in that it also relaxes some of the Ada language rules so that you can omit some implementation details.

If you specify the COMMENTS option, and also specify the /ANALYSIS_DATA qualifier, comments are processed for program design information. You can use the resulting analysis data file with SCA.

The following is a very brief example of using pseudocode placeholders and comment tags to express the design of a procedure. The comment block has several tags, including PACKAGE DESCRIPTION, FUNCTIONAL DESCRIPTION and FORMAL PARAMETERS. The PACKAGE DESCRIPTION and FUNCTIONAL DESCRIPTION tags are text tags, so they contain ordinary text. The FORMAL PARAMETERS tag is a structured tag, so it contains a subtag; namely, the name of the parameter, and the text associated with the subtag.

Note that the package specification defines several subprograms, but that the package body defines the body of only one of those subprograms. Implementation details such as subprogram bodies may be omitted during a design check.

The body of the procedure contains pseudocode, which is a mixture of pure code and pseudocode placeholders. The parts that are real code, such as the **if** statement in the following example, must conform to the regular syntax of the language. The pseudocode placeholders may contain arbitrary text (except for a closing pseudocode delimiter), but must be written on a single line. Great flexibility is available through appropriate mixtures of real code and pseudocode placeholders, as shown in the following example:

```
package PAYROLL is
-- ++
--
-- PACKAGE DESCRIPTION:
--
--      Contains various payroll tasks.
--
-- KEYWORDS:
--
--      {tbs}
--
-- --
    procedure ADD_EMPLOYEE ([formal_part]);
    procedure DELETE_EMPLOYEE ([formal_part]);
    procedure PRINT_EMPLOYEE ([formal_part]);
    procedure READ_EMPLOYEE ([formal_part]);
    procedure PRINT_PAYCHECK  ([formal_part]);
    -- ++
    --
    -- FUNCTIONAL DESCRIPTION:
    --
    --      Compute the amount of an employee's salary and print
    --      a paycheck.
    --
    -- FORMAL PARAMETERS:
    --
    --     NAME:
    --         employee's name
    --
    -- --
    procedure PAYCHECK (NAME : in STRING);
    [basic_declarative_item]...
end PAYROLL;
```

```
package body PAYROLL is
-- ++
--
-- DESIGN:
--
--      {tbs}
--
-- --
    procedure PAYCHECK (NAME : in STRING) is
    -- ++
    --
    -- FUNCTIONAL DESCRIPTION:
    --
    --      Compute the amount of an employee's salary and print
    --      a paycheck.
    --
    -- --
        NORMAL_PAY, OVER_PAY, WEEKLY_PAY : {subtype_indication};
        [basic_declarative_item]...
    begin
        «Fetch the employee's record»

        -- Compute paychecks differently for salaried and
        -- hourly employees
        --
        if «employee is salaried» then
            «Use a fixed weekly salary from employee's record.»
        else
            «Find number of regular and overtime hours worked.»
            «Compute weekly pay.»
        end if;
        «Print the paycheck.»
    end PAYCHECK;
end PAYROLL;
```

D.2 Program Processing

DEC Ada provides three levels of program processing:

- Syntax checking

  Ada compilation units are checked only for correct syntax. For each unit that is syntax checked without error, the program library is updated with information about that unit. However, units added to the program library are considered to be obsolete with respect to design-checked or fully compiled units and must subsequently be fully compiled.

  Units are syntax checked when you enter the ACS LOAD command. Units are also syntax-checked when you specify the /SYNTAX_ONLY qualifier with the DCL ADA, ACS COMPILE or RECOMPILE commands.

- Design checking

  Ada compilation units are syntax checked and design checked. Design checking means that your program is checked for internal consistency, and pseudocode and regular placeholders are accepted (in well-defined contexts) as valid program syntax. Design checking differs from full compilation in that it also relaxes some of the Ada language rules so that you can omit some implementation details.

  For each unit that is design checked without error, the program library is updated with information about that unit. Design-checked units are considered to be obsolete in operations that require full compilation (for example, the ACS LINK command). They must subsequently be fully compiled before you can perform those operations.

  Units are design checked when you use the /DESIGN=PLACEHOLDERS qualifier with the DCL ADA, ACS COMPILE or RECOMPILE commands.

- Full compilation

  Ada compilation units are syntax checked and other compiler checks are performed in strict observance of Ada language rules. LSE placeholders are not allowed.

  For each unit that is compiled without error, the program library is updated with the object module and other products of compilation.

  Units are fully compiled when you use the default versions of the DCL ADA, ACS COMPILE and RECOMPILE commands (the default versions assume the /NOSYNTAX_ONLY and /NODESIGN qualifiers are in effect).

During design checking, the following allowances are made:

- A unit may include any kind of valid LSE placeholder, including pseudocode placeholders. Placeholders are interpreted liberally as representing any construct required at that position by the language rules. A placeholder can be used as an expression; in that case, its type is allowed to match against any type.
- The unit may depend on another unit that was design checked or compiled. A unit that is fully compiled (that is, compiled with the /NODESIGN or the /DESIGN=NOPLACEHOLDERS qualifier in effect) is not allowed to depend on another unit that was design checked.
- Certain omissions are allowed during design checking. In general, these omissions are cases where the body of a program construct is omitted or not fully specified. For example:
  - Incomplete type declarations need not have full type declarations.

  - Private type declarations need not have full type declarations.
  - Subprogram specifications need not have bodies.
  - Package specifications need not have bodies.
  - Task declarations need not have bodies.
  - Task entries need not have corresponding accept statements.
  - Deferred constants need not have full declarations.

In addition, design checking relaxes the processing of Ada source code, as follows:

- No object code is produced.
- Full type declarations are not fully analyzed, so that types are not laid out, all of the values of a type are not accounted for in a case statement, and so on.
- The rules governing visibility, overloading, and conformance checking are relaxed.

D.3 Restrictions on Placeholders

During design checking, placeholders that appear in the Ada template definitions supplied with LSE are generally accepted. However, note the following restrictions:

- To be visible, a declaration must have a valid Ada name, not a placeholder name. You may declare an entity with a placeholder name, but other entities or expressions cannot refer to that declaration. Some specific restrictions result from this rule. For example, a placeholder cannot be used as:
  - The name of a library unit or subunit
  - The parent name in a separate clause
  - The name of a package body or task body
  - The name of a stub
  - The name in a named association
  - A label

- The following placeholders that result from the expansion of LSE templates are allowed in your source code during design processing. However, you cannot replace them with pseudocode placeholders, and they are not reported to SCA as placeholders.
  - [constant] (in object declarations)
  - [constraint]
  - [context_clause]
  - [limited] (in private type declarations)
  - [mode] (of subprogram and entry formal parameters)
  - [reverse] (in a 'for' loop)
  - [type] (in a task type declaration)
  - [use_clause]
  - [with_clause]
- A type declaration with a placeholder as the type definition cannot have a discriminant part, regardless of whether or not the discriminant itself has any placeholders.
- A type mark in a qualified expression cannot be a placeholder.
- A placeholder cannot be used as a variant list in a variant record declaration; also, a placeholder cannot be used as an alternative in a case statement or as an alternative in a select statement.
- A placeholder cannot be used within a character literal. For example, a declaration of the following form is not allowed:

  ```
  type T is ('{graphic_character}',[enumeration_literal],...)
  ```

- A placeholder is not allowed as an operator name in an expression.
- Enumeration representation clauses of the following form are not supported:

  ```
  for ENUM_TYPE use {aggregate};
  ```

  However, once the {aggregate} placeholder is expanded, the representation clause is accepted.
- A placeholder representing a generic formal parameter is only supported if the parameter is the last parameter in the parameter list.
- A placeholder cannot be used as an attribute designator.

- Pragmas containing placeholders can be specified, but have no effect.

D.4 Name Resolution

In general, using placeholders does not change the way that name resolution works for DEC Ada. For example, any nonplaceholder name that is used as a reference must resolve against a previous declaration of the same name; any subprogram call that has a nonplaceholder name must match exactly one previously declared subprogram of a matching name and signature; and so on.

Some variations from the typical Ada resolution are described in this section.

If a type is defined with a placeholder type indication, as in the following example, then all predefined operators — numeric, logical, and catenation — are defined for that type:

```
type T1 is {type_mark};
type T2 is new {type_mark};
```

You can use objects declared to be of the types T1 and T2 in expressions with any of the predefined operators.

If an object is declared to have a placeholder subtype indication (for example, `X: «some type»;`), the declaration is interpreted to mean that the object might be of any type. An object that is itself a placeholder (for example, an actual parameter in a subprogram call, as in `F(«place»);`) is similarly interpreted as possibly being of any type.

Most subprogram calls allow placeholders in the parameter list as long as there is enough information to resolve the call from either the other parameters, the result types, or the number of parameters. If there are multiple possible resolutions of a subprogram call, it will not resolve. For example, consider the following declarations:

```
type T1 is {type_mark};
function F (A, B: INTEGER) return INTEGER;
function F (A, B: T1) return T1;
```

The following statements will not resolve:

```
«place» := F(«place», «place»);
«place» := F(1, 2);
```

However, calls to the predefined operators are interpreted more liberally. The following statement is allowed:

```
«value 1» := «expression 1» + «expression 2»;
```

In this case, there are multiple plus (+) operators available, all of which match the expressions' context. For the predefined operators, the resolution rules have been relaxed during design checking—if the parameters are placeholders or have placeholder type indications, then multiple possible matches of the operator are allowed. This relaxation of the resolution rules applies only to the predefined operators.

If analysis data reporting is requested for this example, no call to plus operators (+) is reported to SCA, as the context cannot be used to determine which plus operator should apply.

Two subprograms are considered to have the same signature if they have the same number of parameters with the same subtype indications. A placeholder subtype indication matches other placeholder subtype indications, regardless of the spelling. For example, the following program would be in error, as the three procedure specifications for the name P1 cannot be distinguished based on the number and type of their parameters; each has one parameter of an unspecified type:

```
procedure P is
   procedure P1 ({identifier}... : {type_mark});
   procedure P1 ([formal_part]);
   procedure P1 ({identifier}... : «type»);
begin
   {statement};
end P;
```

Because the spelling of the type placeholder indications in the signature is not compared, the following association is allowed:

```
procedure P2 ({identifier}... : {type_mark});
procedure P2 ({identifier}... : «subtype indication») is
begin
   null;
end;
```

In general, placeholder spelling is not considered to matter during design checking. There are, however, two exceptions, both involving conformance checking. If a placeholder is used as the name of a parameter in a subprogram, then the spelling of the parameter name must match in both the specification and body, as it does for P2's parameter in the previous example.

In addition, if a placeholder is used as the name of a declared item, such as a subprogram, which has a name after the 'end' literal, then the spelling of the names at the start and end must match. For example, the following is allowed:

```
procedure {procedure_identifier} is
begin
   null;
end [procedure_identifier];
```

D.5 Design Qualifiers

To process an input file as a detailed program design, use the DCL ADA command with one of the following qualifiers:

[NO]COMMENTS	Determines whether comments are processed for program design information. For the COMMENTS option to have effect, you must specify the /ANALYSIS_DATA qualifier with the ADA command. See *Guide to Source Code Analyzer for VMS Systems* for more information on using the Source Code Analyzer (SCA). If you specify NOCOMMENTS, comments are ignored. On AXP systems, the /DESIGN=COMMENTS qualifier is accepted, but has no effect.
[NO]PLACEHOLDERS	Determines whether design checking is performed. If you specify PLACEHOLDERS, compilation units are design checked—LSE placeholders are allowed and some of the Ada language rules are relaxed so that you can omit some implementation details. If you specify NOPLACEHOLDERS, full compilation is done—the compiler is invoked, LSE placeholders are not allowed, and Ada language rules are not relaxed. Note that when you specify this option with the /SYNTAX_ONLY qualifier, it determines only whether LSE placeholders are allowed. If you specify NOPLACEHOLDERS, then only valid Ada syntax is allowed.

If you specify the /DESIGN qualifier without supplying any options, the effect is the same as the following default:

```
/DESIGN=(COMMENTS,PLACEHOLDERS)
```

If you specify only one of the options with the /DESIGN qualifier, the default value for the other option is used. For example, /DESIGN=NOCOMMENTS is equivalent to /DESIGN=(NOCOMMENTS,PLACEHOLDERS). In this case, both qualifiers specify that the unit is design-checked, but comment information is not collected. Similarly, /DESIGN=NOPLACEHOLDERS is equivalent to /DESIGN=(COMMENTS,NOPLACEHOLDERS). In this case, both qualifiers specify that comment information is collected, but the unit is not design-checked (that is, in the absence of the /SYNTAX_ONLY qualifier, units are fully compiled).

Regardless of whether you used the DCL ADA, ACS COMPILE or RECOMPILE command, for each unit that is design checked without error, the program library is updated with information about that unit. Design-checked units are considered to be obsolete in operations that require full compilation. For example, before you link using the ACS LINK command, units must subsequently be recompiled.

When the ACS LOAD command is used in combination with the /SYNTAX_ONLY qualifier, for each unit that is accepted, the program library is updated with information about that unit. Units are syntax checked only, and must be recompiled before you can perform other operations on them.

To enable both design checking and comment processing, include both option parameters on the command line. For example:

```
$ ADA/DESIGN=(PLACEHOLDERS,COMMENTS)/ANALYSIS_DATA HOTEL
```

By default, the DCL ADA, ACS COMPILE and RECOMPILE commands fully compile or recompile the appropriate input file or units. Also, the ACS LOAD command processes the input file as valid Ada source code.

The following lists several qualifier combinations are useful:

- ADA/DESIGN=(PLACEHOLDERS,NOCOMMENTS)

  Useful if you want to check your program for design-level consistency.

- ADA/DESIGN=(NOPLACEHOLDERS,COMMENTS)/ANALYSIS_DATA

  Useful if you want to fully compile your program, but also get SCA information for static analysis, call trees, and report writing.

- ADA/DESIGN/LOAD

  Useful if you want to load files into your program library and also allow placeholders.

D.6 Processing Level Qualifiers

To specify the processing level, use the /PROCESSING_LEVEL qualifier with the ACS CHECK and ACS SHOW PROGRAM commands.

For the ACS CHECK command, these qualifiers determine the kind of obsolete units identified; for the ACS SHOW PROGRAM command, they identify obsolete units in the execution closure of the specified units. For both commands, obsolete units are identified based on the level of processing applied to the unit—syntax checking, design checking, or full compilation. You can request the following qualifiers:

SYNTAX	Determines whether a unit is obsolete because it has been syntax-checked only. Because all units in a program library are at least syntax-checked, and because syntax-checking does not require any particular order of compilation, generally accepts all units as being current.
DESIGN	Determines whether a unit is obsolete because it has been design-checked only. Accepts design-checked units and fully compiled units as being current, unless they are otherwise obsolete (for example, they depend on units that have been syntax-checked only, or they depend on other obsolete units).
FULL	Determines three kinds of obsolete units: units that are obsolete because they have been syntax checked only, units that have been design checked, and units that are obsolete as a result of the compilation of the units they depend on. Units that depend on obsolete units are also considered to be obsolete.

By default, all units are fully checked (/PROCESSING_LEVEL=FULL), and all obsolete units are identified.

E

Diagnostic Messages

This appendix presents information about DEC Ada diagnostic messages, which are generated by the compiler, program library manager, and Ada run-time library. Section E.1 provides information on how diagnostic messages are formatted. Section E.4 lists the Ada run-time diagnostic messages.

Note that the compiler and program library manager messages are intended to be self-explanatory and are not listed in this appendix.

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 *OpenVMS DCL Dictionary* or *OpenVMS User's Manual*).

E.1 Diagnostic Message Format

The general format of a DEC Ada diagnostic message is as follows:

```
%Facility_code-Severity_code-Ident-Message_text
```

Facility_code
Is a three- or four-letter code that identifies a DEC Ada message from the compiler (ADAC), program library manager (ACS), or run-time library (ADA).

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 E.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.

E.2 Diagnostic Messages and Their Severity

A DEC 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 DEC 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 E.3 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 DEC 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  . . .
```

E.3 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 DEC 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
```

E.4 Run-Time Diagnostic Messages

ALICOLILL, Requested alignment for a collection is illegal

Fatal. The DEC 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 DEC 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 DEC 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 DEC 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 DEC 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 DEC 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 DEC 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 DEC 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 DEC 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, DEC 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 DEC Ada); underflow can be handled as an imported 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 DEC 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.

DEC 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.

DEC 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 DEC 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 DEC 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 DEC 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 DEC 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 DEC Ada run-time library is corrupted

Fatal. The data corruption may have been caused by a DEC 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 that is not treated as being equivalent to an Ada predefined 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, DEC 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 DEC 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 DEC 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 DEC 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.

SIZCOLCRE, Attempting to get the size of a collection before its creation

Fatal. This error can occur when you use the 'STORAGE_SIZE attribute to obtain the size of a collection for an access type whose designated type is an incomplete type, and the corresponding full type declaration is not in the same compilation unit.

User Action. Try obtaining the collection size after the full type declaration has been elaborated. See also PROGRAM_ERROR.

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 DEC 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.

UNLCKNOTOWN, A task tried to unlock the global lock without first locking it

Fatal. A task routine in package SYNCHRONIZE_NONREENTRANT_ACCESS tried to unlock the global lock without having first locked it.

User Action. Check the source code for the task to determine why it is trying to unlock the global lock when it does not have it locked. 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 DEC 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.

F

Reporting Problems

If an error occurs while you are using DEC Ada and you believe that the error is caused by a problem with DEC Ada, take one of the following actions:

- If you purchased DEC 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 DEC 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 DEC 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.
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