VAX Ada Run-Time Reference Manual

Order Number: AA–EF88B–TE

May 1989

This manual describes implementation details of VAX Ada in the context of the VMS operating system. It contains information on input-output, representation of types and objects, mixed-language programming, calling VMS system services, exception handling, tasking, and increasing program efficiency. It also lists all of the VAX Ada predefined packages and explains where and how to find the package specifications.

Revision/Update Information: This revision supersedes the *VAX Ada Programmer's Run-Time Reference Manual* (Order No. AA–EF88A–TE)

Operating System and Version: VMS Version 5.0 or higher

Software Version: VAX Ada Version 2.0

THIS PRODUCT CONFORMS TO ANSI/MIL-STD-1815A AS DETERMINED BY THE AJPO UNDER ITS CURRENT TESTING PROCEDURES

digital equipment corporation
maynard, massachusetts

February 1985
Revised, May 1989

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.

© Digital Equipment Corporation 1985, 1989.

All Rights Reserved.
Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation:

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

ZK–3293

Contents

Preface xv

New and Changed Features xix

Chapter 1 Introduction

Chapter 2 Object Representation and Storage

2.1 Type and Object Representations 2–2
2.1.1 Enumeration Types and Objects 2–3
2.1.2 Integer Types and Objects 2–4
2.1.3 Floating-Point Types and Objects 2–5
2.1.3.1 Pragma LONG_FLOAT 2–9
2.1.3.2 VAX F_floating Representation 2–10
2.1.3.3 VAX D_floating Representation 2–11
2.1.3.4 VAX G_floating Representation 2–13
2.1.3.5 VAX H_floating Representation 2–14
2.1.4 Fixed-Point Types and Objects 2–16
2.1.5 Array Types and Objects 2–18
2.1.6 Record Types and Objects 2–18
2.1.7 Access Types and Objects 2–22
2.1.8 Address Types and Objects 2–23
2.1.9 Task Types and Objects 2–24

2.2 Data Optimization 2–24
2.2.1 Pragma PACK 2–24
2.2.2 Length Representation Clauses 2–28
2.2.3 Enumeration Representation Clauses 2–29
2.2.4 Record Representation Clauses 2–29
2.2.5 Alignment Clauses 2–33

2.2.6 Address Clauses 2–35
2.2.7 Determining the Sizes of Types and Objects 2–37

2.3 Storage Allocation and Deallocation 2–41
2.3.1 Storage Allocation 2–42
2.3.2 Storage Deallocation 2–43

Chapter 3 Input-Output Facilities

3.1 Files and File Access 3–2
3.1.1 Ada Sequential Files 3–4
3.1.2 Ada Direct Files 3–4
3.1.3 Ada Relative Files 3–4
3.1.4 Ada Indexed Files 3–5
3.1.5 Ada Text Files 3–5

3.2 Naming External Files 3–6
3.2.1 File Specification Syntax 3–7
3.2.2 Logical Names 3–8

3.3 Specifying External File Attributes 3–11
3.3.1 The VMS File Definition Language (FDL): Primary and Secondary Attributes 3–12
3.3.2 Creation-Time and Run-Time Attributes 3–34
3.3.3 Default External File Attributes 3–35

3.4 File Sharing 3–36

3.5 Record Locking 3–39

3.6 Binary Input-Output 3–40
3.6.1 Sequential File Input-Output 3–44
3.6.2 Direct File Input-Output 3–47
3.6.3 Relative File Input-Output 3–51
3.6.4 Indexed File Input-Output 3–55

3.7 Text Input-Output 3–64
3.7.1 Using the Package TEXT_IO for Terminal Input-Output 3–66
3.7.1.1 Line-Oriented Method 3–69
3.7.1.2 Data-Oriented Method 3–72
3.7.1.3 Mixed Method 3–74
3.7.1.4 Flexible Method 3–75
3.7.2 Line Terminators, Page Terminators, and File Terminators 3–77
3.7.3 Text Input-Output Buffering 3–80

3.7.4 TEXT_IO Carriage Control 3–81
3.7.5 Predefined Instantiations of TEXT_IO Packages 3–85

3.8 Input-Output and Exception Handling 3–86

3.9 Input-Output and Tasking 3–86
3.9.1 Synchronization of Input-Output Operations 3–87
3.9.2 Task Wait States Caused by Input-Output Operations 3–87

Chapter 4 Exception Handling

4.1 Relationship Between Ada Exception Handling and VAX Condition Handling 4–1
4.1.1 Naming and Encoding Ada Exceptions 4–5
4.1.2 Copying Exception Signal Arguments 4–6
4.1.3 The Matching of Ada Exceptions and System-Defined VAX Conditions 4–7

4.2 Making the Best Use of Ada Exception Handling 4–8

4.3 Suppressing Checks 4–9

4.4 Mixed-Language Exception Handling 4–11
4.4.1 Importing Exceptions 4–12
4.4.2 Exporting Exceptions 4–14
4.4.3 The Exception Choice NON_ADA_ERROR 4–15
4.4.4 Signaling VAX Conditions 4–15
4.4.5 Effects of Handling VAX Conditions from an Ada Program . . . 4–21
4.4.6 Fault Handlers 4–25

4.5 Exceptions and Tasking 4–27

Chapter 5 Mixed-Language Programming

5.1 Calling External Routines from Ada Subprograms 5–2

5.2 Calling Ada Subprograms from External Routines 5–4

5.3 Conventions for Passing Data in Mixed-Language Programs 5–6
5.3.1 Ada Semantics 5–7
5.3.2 VAX Calling Standard Conventions 5–8
5.3.2.1 The Call Stack 5–8
5.3.2.2 The Argument List 5–12
5.3.2.3 Parameter-Passing Mechanisms 5–12
5.3.2.4 Function Return 5–13
5.3.2.5 The Call Frame and Register Usage 5–14
5.3.3 VAX Ada Linkage Conventions 5–15
5.3.4 The Importance of Data Representation 5–16

5.4 VAX Ada Default Parameter-Passing Mechanisms 5–18
5.4.1 Scalar Type Parameters 5–18
5.4.2 Array Type Parameters 5–18
5.4.3 Record Type Parameters 5–20
5.4.4 Access Type Parameters 5–20
5.4.5 Address Type Parameters 5–20
5.4.6 Task Type Parameters 5–21
5.4.7 Subprogram Parameters 5–21
5.4.8 Entry Parameters 5–21
5.4.9 VAX Ada Equivalents for VAX Data Types 5–21

5.5 VAX Ada Default Function Return Mechanisms 5–24
5.5.1 Scalar, Access, Address, and Task Type Results 5–25
5.5.2 Array Type Results 5–25
5.5.3 Record Type Results 5–27

5.6 Controlling the Mechanisms for Imported Subprogram Parameters . . 5–28
5.6.1 The VALUE Mechanism Option 5–28
5.6.2 The REFERENCE Mechanism Option 5–29
5.6.3 The DESCRIPTOR Mechanism Option 5–30

5.7 Controlling the Return Mechanisms for Imported Function Results . . 5–35
5.7.1 The VALUE Mechanism Option 5–36
5.7.2 The REFERENCE Mechanism Option 5–36
5.7.3 The DESCRIPTOR Mechanism Option 5–36

5.8 Passing Parameters by Descriptor to Exported Subprograms 5–37

5.9 Sharing Storage with Non-Ada Routines 5–38

Chapter 6 Calling System or Other Callable Routines

6.1 **Using the VAX Ada System-Routine Packages** 6–3
6.1.1 Parameter Types 6–3
6.1.2 Parameter-Passing Mechanisms 6–6
6.1.3 Naming Conventions 6–7
6.1.4 Record Type Declarations 6–7
6.1.5 Default and Optional Parameters 6–11
6.1.6 Calling Asynchronous System Services 6–17
6.1.7 Calling Mathematical Routines 6–17

6.2 **Writing Your Own Routine Interfaces** 6–19
6.2.1 Parameter Types 6–21
6.2.2 Determining the Kind of Call 6–21
6.2.3 Determining the Access Method 6–23
6.2.4 Passing Parameters 6–24
6.2.5 Passing Routines or Subprograms as Parameters 6–24
6.2.6 Default and Optional Parameters 6–24

6.3 **Obtaining Symbol Definitions** 6–25

6.4 **Testing Return Condition Values** 6–26

6.5 **VMS Routine Examples** 6–29

Chapter 7 Using the VAX Common Data Dictionary

7.1 **Using the VAX Ada-from-CDD Translator Utility** 7–2

7.2 **Equivalent VAX Ada and CDDL Data Types** 7–3

7.3 **Example of Using the Ada-from-CDD Translator** 7–5

Chapter 8 Tasking

8.1 **Introduction to Using Ada Tasks on the VMS Operating System** 8–1

8.2 **Task Storage Allocation** 8–7
8.2.1 Storage Created for a Task Object—The Task Control Block 8–8

8.2.2 Storage Created for a Task Activation—The Task Stack 8–9
8.2.2.1 Controlling the Stack Sizes of Task Objects 8–12
8.2.2.2 Controlling the Size of a Main Task Stack 8–13
8.2.3 Stack Overflow and Non-Ada Code 8–14

8.3 Task Switching and Scheduling 8–15

8.4 Special Tasking Considerations 8–17
8.4.1 Deadlock 8–17
8.4.2 Busy Waiting and Non-Ada Code 8–22
8.4.3 Tentative Rendezvous 8–23
8.4.4 Using Delay Statements 8–24
8.4.5 Using Abort Statements 8–24
8.4.6 Interrupting Your Program with CTRL/Y 8–25
8.4.7 Using Shared Variables 8–27
8.4.8 Reentrancy 8–31
8.4.8.1 Reentrancy in Mixed-Language Tasking Programs 8–31
8.4.8.2 Avoiding Nonreentrancy 8–32

8.5 Calling VMS System Service Routines from Tasks 8–35
8.5.1 Effects of System Service Calls on Tasks 8–35
8.5.2 System Services Requiring Special Care 8–37

8.6 Handling Asynchronous System Traps (ASTs) 8–40
8.6.1 The Pragma AST_ENTRY and the AST_ENTRY Attribute 8–40
8.6.2 Constraints on Handling ASTs 8–43
8.6.3 Calling Ada Subprograms from Non-Ada AST Service Routines 8–43
8.6.4 Examples of Handling ASTs from Ada Programs 8–45

8.7 Measuring and Tuning Tasking Performance 8–48

Chapter 9 Improving Run-Time Performance

9.1 Compiler Optimizations 9–1

9.2 Using the Pragma INLINE 9–3
9.2.1 Explicit Use 9–4
9.2.2 Implicit Use 9–6
9.2.3 Pragma INLINE Examples 9–7
9.2.3.1 Inline Expansion of Subprogram Specifications and Bodies 9–7
9.2.3.2 Inline Expansion of Generic Subprograms 9–9

9.3 **Making Use of Generics** 9–11
9.3.1 Using the Pragma INLINE_GENERIC 9–12
9.3.2 Using the Pragma SHARE_GENERIC 9–14
9.3.3 Library-Level Generic Instantiations 9–17

9.4 **Techniques for Reducing CPU Time and Elapsed Time** 9–19
9.4.1 Decreasing the CPU Time of a VAX Ada Program 9–20
9.4.1.1 Eliminating Run-Time Checks 9–21
9.4.1.2 Reducing Function and Procedure Call Costs 9–22
9.4.1.3 Using Scalar Variables and Avoiding Expensive Operations on Composite Types 9–25
9.4.2 Decreasing the Elapsed Time of a VAX Ada Program 9–27
9.4.2.1 Controlling Paging Behavior 9–28
9.4.2.2 Improving Input-Output Behavior 9–28
9.4.2.3 Overlapping Unrelated Input-Output and Instruction Execution 9–28

Chapter 10 Additional Programming Considerations

10.1 **Working with Address Values** 10–1

10.2 **Using Low-Level System Features** 10–2
10.2.1 The VAX Device and Processor Register and Interlocked Operations 10–3
10.2.2 Unsigned Types in the Package SYSTEM 10–6

10.3 **Working with Varying Strings** 10–9

10.4 **Assigning Array Values** 10–10

10.5 **Sharing Memory Between VAX CPUs** 10–13

Appendix A VAX Ada Predefined Instantiations

Appendix B VAX Ada Packages

B.1 **Packages ASSERT, ASSERT_EXCEPTIONS, and ASSERT_GENERIC** B–10

B.2 **Package CDD_TYPES** B–14

B.3 Package CONDITION_HANDLING B–19

B.4 Package CONTROL_C_INTERCEPTION B–26

B.5 Package MATH_LIB B–26

B.6 Package STARLET B–28

B.7 Package SYSTEM_RUNTIME_TUNING B–39

B.8 Package TASKING_SERVICES B–42

Index

Examples

2–1 Using an Address Clause and LIB$GET_VM 2–36

2–2 Using UNCHECKED_DEALLOCATION to Control Access Type Storage Deallocation 2–44

3–1 Creating and Opening a Relative File for Read Sharing 3–37

3–2 Using a Mixed-Type File 3–42

3–3 Using the Package SEQUENTIAL_IO 3–47

3–4 Using the Package DIRECT_MIXED_IO 3–50

3–5 Using the Package RELATIVE_IO 3–54

3–6 Using the Package INDEXED_IO 3–58

3–7 Using the Package INDEXED_MIXED_IO 3–62

3–8 Using the Package TEXT_IO 3–67

3–9 Example of Line-Oriented TEXT_IO 3–71

3–10 Example of Data-Oriented TEXT_IO 3–73

3–11 Example of Flexible TEXT_IO 3–75

4–1 Use of Pragma SUPPRESS_ALL 4–10

4–2 Handling SYS$GETJPIW Status Values as Ada Exceptions 4–16

4–3 Handling SYS$GETJPIW Status Values as VMS Conditions 4–19

6–1 Calling SYS$TRNLNM Using the Package STARLET 6–29

6–2 Calling SYS$GETQUI Using the Package STARLET 6–31

6–3 Calling SYS$CRMPSC Using the Package STARLET 6–34

6–4 Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB 6–36

6–5 Calling SMG Routines Using the Package SMG 6–40

6–6 Calling SYS$TRNLNM Using an Import Pragma 6–43
6–7 Using SYSTEM.IMPORT_VALUE to Obtain a Global Symbol Value 6–46
8–1 Interactive Array Sort Using Tasks 8–3
8–2 Leaving a Master to Release a Task Control Block 8–10
8–3 Controlling the Size of a Task's Stack 8–13
8–4 An Exception-Induced Deadlock 8–19
8–5 A Self-Calling Deadlock 8–20
8–6 A Circular-Calling Deadlock 8–21
8–7 A Dynamic-Circular-Calling Deadlock 8–22
8–8 A Nonreentrant Subprogram 8–32
8–9 A Reentrant Subprogram 8–33
8–10 Using a Serializing Task to Prevent Reentry 8–34
8–11 Deadlock Caused by a Call to SYS$SETAST 8–38
8–12 Unpredictability of SYS$EXIT 8–39
8–13 Simple Use of the Pragma AST_ENTRY and the AST_ENTRY Attribute 8–45
8–14 Using an AST Entry to Intercept a CTRL/C 8–46
10–1 One Use of the Interlocked Queue Operations 10–4
10–2 Sharing Memory Between Two or More Programs Running on One or More VAX CPUs 10–13

Figures

2–1 F_floating Representation 2–10
2–2 D_floating Representation 2–12
2–3 G_floating Representation 2–13
2–4 H_floating Representation 2–15
3–1 Using a Mixed-Type File 3–43
3–2 Using a Uniform-Type File 3–44
3–3 An Ada Text File, Showing Line, Page, and File Terminators 3–78
4–1 Execution of a FORTRAN Program with FOR$UNDERFLOW_HANDLER 4–23
4–2 The Effect of an Ada Procedure Containing an Others Handler 4–24
4–3 FOR$UNDERFLOW_HANDLER Established for a FORTRAN Subroutine 4–26
5–1 A Call Stack at Run Time 5–10
5–2 An Argument List 5–12
5–3 A Call Stack 5–15
5–4 Area Control Block Used in Returning Some Function Results 5–26

Tables

2–1 Range of Values and Storage Sizes for VAX Ada Predefined Integer Types 2–5
2–2 VAX Type Representations and Storage Sizes for VAX Ada Predefined Floating-Point Types 2–6
2–3 Model Numbers Defined for Each Floating-Point Type 2–8
2–4 Safe Numbers Defined for Each Floating-Point Type 2–9
2–5 Packable Types 2–25
2–6 Effects of Packing the Components of Arrays and Records 2–26
2–7 Comparison of SIZE and MACHINE_SIZE Attribute Results 2–38
2–8 Results of Size Attributes for Various Types and Objects 2–40
3–1 Predefined (Default) Logical Names 3–9
3–2 Equivalence Strings for Default Logical Names for Process-Permanent Files 3–11
3–3 FDL Primary and Secondary Attribute Descriptions 3–12
3–4 Commonly Used FDL Attributes 3–20
3–5 SEQUENTIAL_IO: Default File Attributes 3–45
3–6 SEQUENTIAL_MIXED_IO: Default File Attributes 3–46
3–7 DIRECT_IO: Default File Attributes 3–48
3–8 DIRECT_MIXED_IO: Default File Attributes 3–49
3–9 RELATIVE_IO: Default File Attributes 3–52
3–10 RELATIVE_MIXED_IO: Default File Attributes 3–52
3–11 INDEXED_IO: Default File Attributes 3–56
3–12 INDEXED_MIXED_IO: Default File Attributes 3–56
3–13 TEXT_IO: Default File Attributes 3–65
3–14 VAX Ada Carriage-Control Options 3–82
3–15 FORTRAN Carriage-Control Characters 3–84
4–1 Relationship Between Ada Exception Handling and the CHF 4–3
4–2 Ada Predefined Exceptions 4–5
4–3 VAX Conditions that Match Ada Exceptions 4–8
4–4 Run-Time Checks and Their Corresponding Predefined Exceptions 4–9
5–1 VAX Registers 5–14
5–2 Default Descriptor Classes Used by VAX Ada for Array Parameter Passing 5–19
5–3 VAX Ada Equivalents for VAX Data Types and Their Valid Passing Mechanisms in VAX Ada 5–22

5–4 Default Descriptor Class Names Used for the DESCRIPTOR Mechanism 5–32
5–5 Type Requirements for Descriptor Classes Used by VAX Ada in Importing Routines 5–32
5–6 Descriptor Data Types Used 5–35
5–7 Program Section Properties 5–39
6–1 VMS Data Structures 6–4
6–2 VAX Ada Equivalents for VMS Access Methods 6–23
7–1 Equivalent CDD and VAX Ada Data Types 7–4
9–1 Comparison of the Effects of the Pragmas INLINE_GENERIC and SHARE_GENERIC 9–12
B–1 VAX Ada Predefined Packages B–2

Preface

This manual describes implementation details of VAX Ada in the context of the VMS operating system. It contains information on input-output, representation of types and objects, mixed-language programming, calling VMS system services, exception handling, tasking, and increasing program efficiency. It also lists and gives the specifications for some of the VAX Ada predefined packages.

Intended Audience

This manual is intended primarily for systems and applications programmers, or any other programmers whose work requires the use of operating system features outside of the language, advanced Ada features, or more than one VAX language. The reader should have a working knowledge of Ada and some familiarity with the VMS operating system.

Structure of This Document

This manual has ten chapters and two appendixes:

- Chapter 1 introduces VAX Ada.
- Chapter 2 explains how VAX Ada objects and types are represented and sized; it also gives information on sharing object storage among Ada and non-Ada routines.
- Chapter 3 discusses VAX Ada input-output, giving details about file sharing, record locking, and the VAX Ada input-output packages. This chapter also summarizes information about the VMS File Definition Language and the specification of file names.

- Chapter 4 describes the implementation of VAX Ada exception handling and discusses the importing and exporting of VAX conditions and Ada exceptions.
- Chapter 5 describes the VAX Ada parameter-passing mechanisms and import-export pragmas, and discusses how to write mixed-language programs that involve VAX Ada.
- Chapter 6 explains how to call system and other callable routines (VMS system services, Run-Time Library routines, and so on).
- Chapter 7 describes how to access the VAX Common Data Dictionary from VAX Ada.
- Chapter 8 discusses tasking issues, including issues related to calling non-Ada routines (such as VMS system services) from tasks.
- Chapter 9 gives information on how to make VAX Ada programs more efficient.
- Chapter 10 discusses additional details of VAX Ada that you need to consider when writing VAX Ada programs.
- Appendix A lists all of the VAX Ada predefined generic instantiations.
- Appendix B lists all of the VAX Ada packages, and gives the specifications for the packages that are system-specific or that do not have their specifications given in the *VAX Ada Language Reference Manual*.

Associated Documents

The following manuals from the VAX Ada documentation set may be of interest to you:

- The *VAX Ada Language Reference Manual*, which gives information on VAX Ada language details
- *Developing Ada Programs on VMS Systems*, which gives information on how to develop and run VAX Ada programs using the VAX Ada program library manager and VMS Debugger

You should also have access to the VMS system documentation.

The following Ada textbooks may also be of interest:

- Barnes, J.G.P. *Programming in Ada*. Reading, Massachusetts: Addison-Wesley, second edition, 1984.
- Booch, Grady. *Software Components with Ada: Structures, Tools and Subsystems*. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., 1987.

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

Conventions

Convention	Meaning
RETURN	In interactive examples, a label enclosed in a box indicates that you press a key on your keyboard, for example, RETURN.
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1988 11:55:22	Interactive examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
. . .	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
task	Boldface indicates Ada reserved words.
*type*_name	Italicized words in syntax descriptions indicate descriptive prefixes that are intended to give additional semantic information rather than to define a separate syntactic category.

Convention	Meaning
[expression]	Square brackets indicate that the enclosed item is optional.
{, mechanism_name }	Braces indicate that the enclosed item may be repeated zero or more times.
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

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

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

- The implementation of fixed-point types has changed (see Chapter 2).
- Record components may be biased under certain conditions (see Chapter 2).
- Address representation clauses are allowed for variables (see Chapter 2).
- The FDL SHARING PROHIBIT attribute is no longer set by the input-output packages (see Chapter 3).
- Indexed files can be sorted by descending (as well as ascending) keys (see Chapter 3).
- File sharing and record locking are available for all input-output files (see Chapter 3).
- In response to Ada interpretation AI-00387, VAX Ada raises CONSTRAINT_ERROR wherever the standard requires that NUMERIC_ERROR be raised (see Chapter 4).
- Support for the pragma SUPPRESS has been added (see Chapter 4).
- Improvements have been made to the way in which the VAX Ada run-time library deals with unhandled exceptions signaled by non-Ada code. In addition, messages are now displayed before waiting begins for dependent tasks (see Chapter 4).
- Changes have been made to the default passing mechanisms for some parameter and function result types (see Chapter 5).
- The RESULT_MECHANISM mechanism option has been added to the pragma IMPORT_FUNCTION (see Chapter 5).

- The pragma EXPORT_VALUED_PROCEDURE has been added (see Chapters 5 and 6).
- The function SYSTEM.IMPORT_VALUE has been added (see Chapter 6).
- The FIRST_OPTIONAL_PARAMETER mechanism option has been added to the import pragmas (see Chapter 6).
- A set of VMS Run-Time Library (DTK, LIB, MTH, OTS, PPL, SMG, and STR) and utility (CLI, NCS, LBR, and SOR) packages has been added (see Chapter 6).
- The pragma MAIN_STORAGE has been added (see Chapter 8).
- Support for the pragma SHARED has been added (see Chapter 8).
- The effects and restrictions on using the pragma INLINE have been further defined (see Chapter 9).
- The pragmas INLINE_GENERIC and SHARE_GENERIC have been added (see Chapter 9).
- A number of hardware-related types and operations have been added to the package SYSTEM (see Chapter 10).
- The packages SYSTEM_RUNTIME_TUNING, ASSERT_GENERIC, ASSERT_EXCEPTIONS, and the package instantiation ASSERT have been added (see Appendix B).
- The package TASKING_SERVICES now includes an interface to the VMS $GETQUIW system service, and problems with the TASK_ENQW and TASK_UPDSECW procedures have been identified and corrected (see Appendix B).
- Changes and corrections have been made to the package STARLET (see Appendix B for type definition changes).

Chapter 1

Introduction

Ada is a general-purpose programming language suitable for writing large-scale and real-time systems programs. For example, Ada is strongly typed, provides for exact or approximate numerical calculations, supports concurrency, and allows separate compilation of program units. The language is specified in ANSI/MIL-STD-1815A-1983 and ISO/8652-1987, *Reference Manual for the Ada Programming Language*, which has been reproduced, with supplementary Digital insertions, as the *VAX Ada Language Reference Manual*.

VAX Ada implements the ANSI and ISO standard Ada programming language on the VMS operating system. VAX Ada provides for all of the standard language features. VAX Ada also provides additional packages, attributes, and pragmas designed to allow Ada programmers to work efficiently in a VMS environment and make use of the VMS operating system.

Like other languages in the VAX Common Language Environment, VAX Ada has the following properties:

- It conforms to the VAX Procedure Calling Standard.
- It interacts with the VMS Run-Time Library.
- It uses VAX Record Management Services (RMS) to implement input-output.
- It depends on the VAX Condition Handling Facility (CHF) to implement exception handling.

The VAX Ada compiler produces highly optimized object code and makes use of the VAX hardware instruction set.

VAX Ada is described in the following chapters, with a focus on those VAX Ada features that allow you to interact with the VMS operating system and other VAX languages. Machine- and operating-system-related implementation details are provided as appropriate.

By being able to call VMS system, Run-Time Library, and callable utility routines from Ada subprograms or tasks, you can write programs that make efficient use of all of the capabilities of the VMS operating system. By being able to call other languages and handle exceptions from both Ada and non-Ada code, you can make use of existing non-Ada routines, or take advantage of features of other languages that may be suitable for your application.

Chapter 2

Object Representation and Storage

An Ada *object* is an entity that can have values of a particular type. For each Ada object, the VAX Ada compiler determines how much storage is required, where and when that storage will be allocated and deallocated, and how the different values of the object are represented. The compiler makes these determinations based on the type of the object, the subtype of the object, and the use of the object.

In simple cases, the representation and storage of objects is determined at compile time. In more complex cases (such as the case of an array object whose bounds are not computed until run time), the compiler generates code that computes the amount of storage required at run time. In general, the compiler chooses storage sizes and representations that make the best compromise between CPU time and the amount of memory required by the generated code.

Pragmas and representation clauses allow you to control how objects are represented and stored. You most often need this control when you are working with the following kinds of objects:

- Objects whose addresses are explicitly obtained with the ADDRESS attribute
- Objects whose addresses are explicitly specified with an address representation clause
- Objects that are passed to imported routines or used in exported subprograms
- Objects that are imported/exported using the VAX Ada pragma PSECT_OBJECT

When the VAX Ada compiler determines how to store and represent objects, it uses rules that are similar to those used by other VAX language compilers. Thus, you can still use those objects whose storage and representation cannot be controlled in the VAX Common Language Environment.

For example, simple objects of the type STANDARD.BOOLEAN are represented as unsigned bytes containing the values 0 (FALSE) and 1 (TRUE). Similarly, the types STANDARD.CHARACTER and STANDARD.STRING correspond to the VAX notions of character and string, and objects of these types are represented as one or more unsigned bytes (although the type STANDARD.CHARACTER does not include the upper half of the DEC Multinational Character Set).

To increase efficiency, the VAX Ada compiler may use alternative representations for some objects (for example, it may use a 32-bit longword rather than an 8-bit byte for some objects of the type STANDARD.BOOLEAN, as this representation tends to be more efficient in the use of CPU resources). However, the compiler will not choose an alternative representation for objects that are visible outside the Ada program; that is, it will not choose an alternative representation for an object that is passed to an imported routine, passed into an exported subprogram, or imported/exported using the VAX Ada pragma PSECT_OBJECT.

This chapter discusses the following topics:

- The representation and storage chosen by the VAX Ada compiler for objects of a variety of VAX Ada types
- How to tailor the representation of the objects in your program to suit your particular application
- Storage allocation and deallocation

You should be familiar with the material in Chapters 3 and 13 of the *VAX Ada Language Reference Manual* before using the material in this chapter.

2.1 Type and Object Representations

The following sections describe the representations and storage sizes chosen by the VAX Ada compiler for objects of the various Ada type classes, including scalar (enumeration, integer, floating-point, and fixed-point), array, record, access, address, and task types.

2.1.1 Enumeration Types and Objects

Each enumeration literal in an enumeration type has a corresponding internal code. Unless otherwise specified in an enumeration representation clause, the internal codes for an enumeration type are represented by the integers from 0 to N – 1, where N is the number of enumeration literals in the type. For example, the internal codes for the enumeration literals of the Ada predefined types STANDARD.BOOLEAN and STANDARD.CHARACTER are as follows:

Enumeration Type	Internal Codes
STANDARD.BOOLEAN	0 (FALSE) 1 (TRUE)
STANDARD.CHARACTER	0..127[1]

[1]The internal code for each character is its conventional ASCII value (the NUL character has the internal code 0, 'A' has the internal code 65, 'a' has the internal code 97, and so on); see the specification of the package STANDARD in Annex C of the *VAX Ada Language Reference Manual.*

Because Ada does not include the DEC Multinational Character Set in the package STANDARD, the internal codes 128..255 have no meaning in VAX Ada for enumeration literals of the type STANDARD.CHARACTER.

Section 2.2.3 explains how to use an enumeration representation clause to specify other values (including negative ones) for internal codes.

The amount of storage allocated by the VAX Ada compiler for an object of an enumeration type depends on the range of the internal codes and on any length representation clauses that provide a size for the type or first named subtype. (A first named subtype is a subtype declared by a type declaration; see Chapter 13 of the *VAX Ada Language Reference Manual.*) Note that when you specify a length representation clause for a first named subtype, the clause may not be applied to the representation of objects of the base type; for example, this effect may occur with loop parameters.

Thus, for simple enumeration objects and enumeration components of unpacked arrays and records, the VAX Ada compiler chooses a byte (8 bits), a word (16 bits), or a longword (32 bits)—whichever is smallest—to represent an object of an enumeration type. The size chosen is large enough to represent all of the values of the type, and is greater than or equal to any applicable length representation clause.

For most enumeration types, the representation is unsigned; the representation is signed only when the first internal code is negative.

For example:

```
type ANSWER is (YES, NO, UNDECIDED);
```

An object of the type ANSWER will be stored in an unsigned byte, because a byte is all that is needed to represent the default internal codes (0, 1, and 2) corresponding to YES, NO, and UNDECIDED. To guarantee a particular representation or to achieve a signed representation, you can use an enumeration representation clause. See Section 2.2.3 for more information.

2.1.2 Integer Types and Objects

VAX Ada provides three predefined integer types:

SHORT_SHORT_INTEGER
SHORT_INTEGER
INTEGER

These types are declared in the predefined package STANDARD (see Annex C of the *VAX Ada Language Reference Manual*).

Values for objects of all three integer types are represented as signed, two's complement (binary) numbers.

You can achieve an unsigned representation for integer objects by declaring an integer type with a length representation clause (see Section 2.2.2). However, because of the way the Ada language defines integer operations, operations on these unsigned objects will involve signed intermediate values. See Chapter 10 for more information on working with unsigned types.

Table 2–1 lists the range of integer values and storage sizes for each of these predefined integer types.

Table 2–1: Range of Values and Storage Sizes for VAX Ada Predefined Integer Types

Ada Type	Range of Values	Storage Size (Bits)
SHORT_SHORT_INTEGER	$-2^7..2^7 - 1$ –128..127	8
SHORT_INTEGER	$-2^{15}..2^{15} - 1$ –32_768..32_767	16
INTEGER	$-2^{31}..2^{31} - 1$ –2_147_483_648..2_147_483_647	32

2.1.3 Floating-Point Types and Objects

Floating-point types provide approximations to the real numbers, with relative bounds on the errors. For each floating-point type—predefined and nonpredefined—the VAX Ada compiler chooses one of the four VAX floating-point data representations, depending on the required range and accuracy:

F_floating
D_floating
G_floating
H_floating

The chosen representation and size is used for all objects of the type, regardless of the objects' subtypes, and regardless of whether or not the objects are themselves part of packed array or record objects. Sections 2.1.3.2 through 2.1.3.5 explain the VAX floating-point data representations in detail.

VAX Ada provides a number of predefined floating-point types. Table 2–2 lists the representation and storage size for each type.

Table 2–2: VAX Type Representations and Storage Sizes for VAX Ada Predefined Floating-Point Types

Ada Type	VAX Representation	Storage Size (Bits)
Defined in the Package STANDARD:		
FLOAT	F_floating	32
LONG_FLOAT	D_floating or G_floating[1]	64
LONG_LONG_FLOAT	H_floating	128
Defined in the Package SYSTEM:		
F_FLOAT	F_floating	32
D_FLOAT	D_floating	64
G_FLOAT	G_floating	64
H_FLOAT	H_floating	128

[1]By default, or in the presence of the pragma LONG_FLOAT(G_FLOAT), the type LONG_FLOAT has a G_floating representation; it has a D_floating representation in the presence of the pragma LONG_FLOAT(D_FLOAT). Section 2.1.3.1 of this manual and Chapter 3 of the *VAX Ada Language Reference Manual* discuss this pragma in more detail.

You can also use the ACS CREATE LIBRARY, CREATE SUBLIBRARY, and SET PRAGMA commands to control the representation of the type LONG_FLOAT. These commands are described in *Developing Ada Programs on VMS Systems*.

VAX Ada allows you to define your own floating-point types. The choice of representation for nonpredefined floating-point types that are not explicitly derived depends on the precision (**digits**) and the range specified. The VAX Ada compiler chooses the first of the types STANDARD.FLOAT, STANDARD.LONG_FLOAT, and STANDARD.LONG_LONG_FLOAT that has adequate precision and range, and uses it as the parent type from which the new type is derived.

If the G_floating representation of the type LONG_FLOAT is in effect for the compilation (see Table 2–2 and Section 2.1.3.1), the following representations are used if the specified range can also be accommodated:

Digits Specified	G_floating Representations
1 .. 6	F_floating
7 .. 15	G_floating
16 .. 33	H_floating

If the D_floating representation of the type LONG_FLOAT is in effect for the compilation (see Table 2–2 and Section 2.1.3.1), the following representations are used if the specified range can also be accommodated:

Digits Specified	D_floating Representations
1 .. 6	F_floating
7 .. 9	D_floating
10 .. 33	H_floating

For example, the pragma LONG_FLOAT in the following declaration ensures that the D_floating representation of the type LONG_FLOAT is in effect when the declaration is compiled. However, the compiler will choose the type STANDARD.LONG_LONG_FLOAT as the parent type for the type SIZE because although a D_floating representation satisfies the precision, it does not satisfy the range.

```
pragma LONG_FLOAT(D_FLOAT);
package FLOAT_TYPES is
   type SIZE is digits 9 range -0.1E-50 .. 0.1E+50;
    . . .
end FLOAT_TYPES;
```

In all cases, the choice of representation for a floating-point type is determined by the model number limits specified by the Ada language (see Chapter 3 of the *VAX Ada Language Reference Manual*). However, once the representation is chosen, the full accuracy of the underlying VAX floating-point type is used in any calculations involving numbers of that type. For example, the following type declaration causes the full 16 decimal digits of accuracy provided by the VAX D_floating hardware representation to be used in calculations involving objects of the type:

```
type VOLUME is digits 9 range -100.0 .. 100.0;
```

Table 2–3 lists the model numbers for each VAX floating-point type (and thereby for each VAX Ada predefined floating-point type). The ranges in the table are approximate; the exact ranges are listed in Appendix F of the *VAX Ada Language Reference Manual*. You can also find the exact ranges by evaluating language-defined attributes T'SMALL and T'LARGE, where T is the floating-point type. Table 2–3 lists only the positive ranges; all floating-point numbers are, in fact, signed, and an equivalent negative range (as well as zero) exists for each type.

Table 2–3: Model Numbers Defined for Each Floating-Point Type

VAX Ada Types and Representations	Digits (D)	Mantissa Bits (B)	Exponent Range (–4*B..+4*B)	Approximate Range
F_floating F_FLOAT FLOAT	6	21	–84 .. 84	2.5E–26 .. 1.9E+25
D_floating D_FLOAT LONG_FLOAT	9	31	–124 .. 124	2.3E–38 .. 2.1E+37
G_floating G_FLOAT LONG_FLOAT	15	51	–204 .. 204	1.9E–62 .. 2.5E+61
H_floating H_FLOAT LONG_LONG_FLOAT	33	111	–444 .. 444	1.1E–134 .. 4.5E+133

For both predefined and nonpredefined types, the Ada language rules about safe numbers also apply (see Chapter 3 of the *VAX Ada Language Reference Manual*). Table 2–4 lists the safe numbers for each VAX floating-point type (and thereby for each VAX Ada floating-point type). The ranges in the table are approximate; the exact ranges are listed in Appendix F of the *VAX Ada Language Reference Manual*. You can also find the exact ranges by evaluating the language-defined attributes T'SAFE_SMALL and T'SAFE_LARGE, where T is the floating-point type. As with the model numbers, only the positive ranges are listed; each type includes zero and a set of corresponding negative values.

Table 2–4: Safe Numbers Defined for Each Floating-Point Type

VAX Ada Types and Representations	Digits (D)	Mantissa Bits (B)	Exponent Range (–E..+E)	Approximate Range
F_floating F_FLOAT FLOAT	6	21	–127 .. 127	2.9E–39 .. 1.7E+38
D_floating D_FLOAT LONG_FLOAT	9	31	–127 .. 127	2.9E–39 .. 1.7E+38
G_floating G_FLOAT LONG_FLOAT	15	51	–1023 .. 1023	5.5E–309 .. 8.9E+307
H_floating H_FLOAT LONG_LONG_FLOAT	33	111	–16383 .. 16383	8.4E–4933 .. 5.9E+4931

2.1.3.1 Pragma LONG_FLOAT

The VAX Ada predefined pragma LONG_FLOAT acts as a program library switch that controls whether the G_floating or D_floating representation is used to represent the type LONG_FLOAT. Use of this pragma implies a recompilation of the predefined environment—the package STANDARD—for a given program library. See the *VAX Ada Language Reference Manual* for the specific rules governing the use of this pragma; see *Developing Ada Programs on VMS Systems* for a discussion of the implications of recompiling the package STANDARD.

For example, the compilation of the following unit will cause all subsequent compilations in the same library to use the set of representations that include D_floating, as appropriate (see Section 2.1.3):

```
pragma LONG_FLOAT(D_FLOAT);
package USE_D_FLOAT is

   -- D_floating representation will be used.
   --
   type MY_D_FLOAT is digits 9 range -100.0 .. 100.0;

   -- H_floating representation will be used.
   --
   type MY_H_FLOAT is digits 11 range -100.0 .. 100.0;
```

```
   -- D_floating representation will be used.
   --
   D_OBJECT: LONG_FLOAT;
    . . .
end USE_D_FLOAT;
```

To return to G_floating representations, you can use one of the following methods:

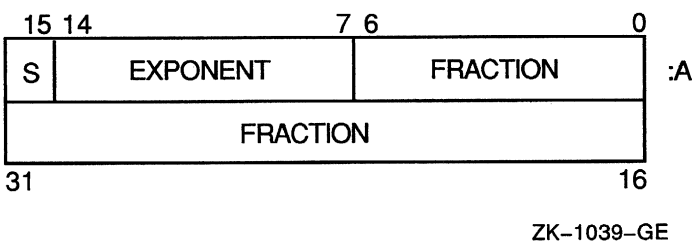
- Compile another unit (in the same library) that contains the pragma LONG_FLOAT(G_FLOAT).
- Use the ACS SET PRAGMA command.
- Recreate your library by first deleting it with either the ACS DELETE LIBRARY or DELETE SUBLIBRARY command, and then creating it with the ACS CREATE LIBRARY or SUBLIBRARY command.

See *Developing Ada Programs on VMS Systems* for information on the ACS commands.

2.1.3.2 VAX F_floating Representation

An F_floating-point number (single precision) is represented in memory by 4 contiguous bytes (32 bits). The bits are numbered from the right, 0 through 31, as shown in Figure 2–1.

Figure 2–1: F_floating Representation



The address of an F_floating-point value is the address of the byte containing bit 0 (address A in Figure 2–1). The form of the value is signed magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 7 are an excess 128 binary exponent.
- Bits 6 through 0 and 31 through 16 are a normalized 23-bit fraction, with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and from 0 through 6.

The 8-bit exponent field encodes the values 0 through 255 as follows:

- An exponent value of 0 with a sign bit of 0 indicates that the floating-point number has a value of 0.
- Exponent values of 1 through 255 indicate binary exponents of –127 through +127.

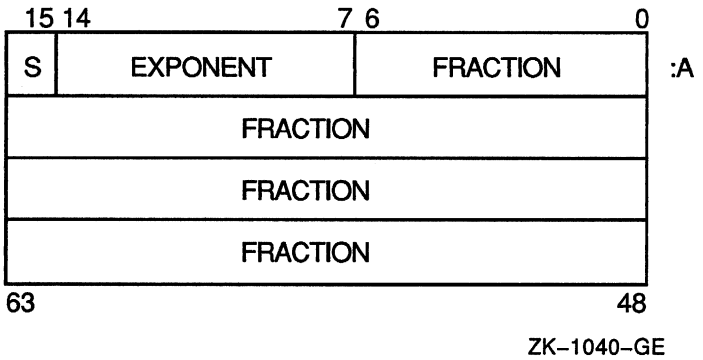
If you are doing unchecked conversions to floating-point types (see Chapter 13 of the *VAX Ada Language Reference Manual*), note that an exponent value of 0 with a sign bit of 1 is considered to be a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

In VAX Ada, the VAX F_floating representation is used to represent the set of model numbers shown in Table 2–3 and the set of safe numbers shown in Table 2–4. On VAX machines, the value of an F_floating-point number is in the approximate range of $0.29*10^{-38}$ through $1.7*10^{38}$. The precision of an F_floating-point value is approximately one part in 2^{23}, or at least 6 decimal digits.

2.1.3.3 VAX D_floating Representation

A D_floating-point number (double precision) is represented in memory by 8 contiguous bytes (64 bits). The bits are numbered from the right, 0 through 63, as shown in Figure 2–2.

Figure 2–2: D_floating Representation



The address of a D_floating-point value is the address of the byte containing bit 0 (address A in Figure 2–2). The form of the value is signed magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 7 are an excess 128 binary exponent.
- Bits 6 through 0 and 63 through 16 are a normalized 59-bit fraction, with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance are numbered 48 through 63, 32 through 47, 16 through 31, and 0 through 6.

The 8-bit exponent field encodes the values 0 through 255 as follows:

- An exponent value of 0 with a sign bit of 0 indicates that the floating-point number has a value of 0.
- Exponent values of 1 through 255 indicate binary exponents of –127 through +127.

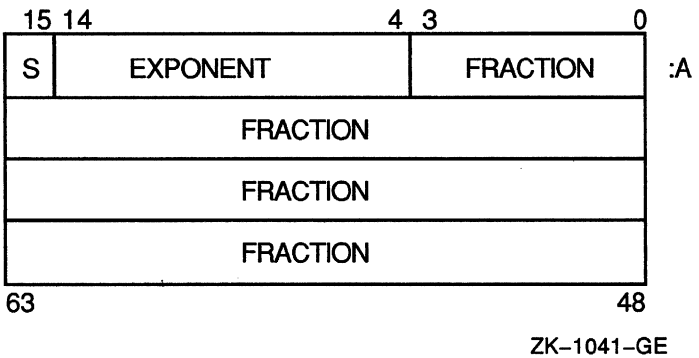
If you are doing unchecked conversions to floating-point types (see Chapter 13 of the *VAX Ada Language Reference Manual*), note that an exponent value of 0 with a sign bit of 1 is considered to be a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

In VAX Ada, the VAX D_floating representation is used to represent the set of model numbers shown in Table 2–3 and the set of safe numbers shown in Table 2–4. On VAX machines, the exponent conventions and approximate range of values are the same for D_floating-point values as for F_floating-point values. However, the precision of a D_floating-point value is approximately one part in 2^{55}, or 16 decimal digits.

2.1.3.4 VAX G_floating Representation

A G_floating-point number (double precision) is represented in memory by 8 contiguous bytes (64 bits). The bits are numbered from the right, 0 through 63, as shown in Figure 2–3.

Figure 2–3: G_floating Representation



The address of a G_floating-point value is the address of the byte containing bit 0 (address A in Figure 2–3). The form of a G_floating-point value is signed magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 4 are an excess 1024 binary exponent.
- Bits 3 through 0 and 63 through 16 represent a normalized 53-bit fraction, with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing signficance go from 48 through 63, 32 through 47, 16 through 31, and 0 through 3.

The 11-bit exponent field encodes the values 0 through 2047 as follows:

- An exponent value of 0 with a sign bit of 0 indicates that the G_floating-point number has a value of 0.
- Exponent values of 1 through 1047 indicate binary exponents of –1023 through +1023.

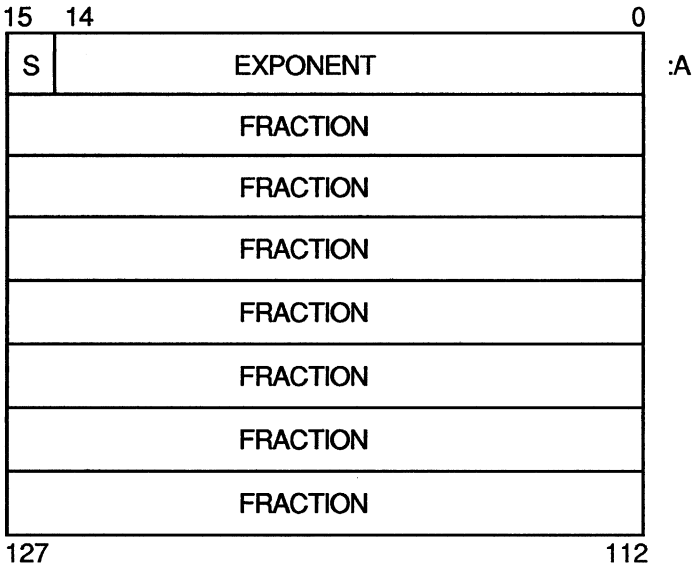
If you are doing unchecked conversions to floating-point types (see Chapter 13 of the *VAX Ada Language Reference Manual*), note that an exponent value of 0 with a sign bit of 1 is considered to be a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

In VAX Ada, the VAX G_floating representation is used to represent the set of model numbers shown in Table 2–3 and the set of safe numbers shown in Table 2–4. On VAX machines, the value of a G_floating-point number is in the approximate range of $0.56*10^{-308}$ through $0.90*10^{308}$. The precision of a G_floating-point value is approximately one part in 2^{52}, or 15 decimal digits.

2.1.3.5 VAX H_floating Representation

An H_floating-point (quadruple precision) value is represented in memory by 16 contiguous bytes (128 bits). The bits are numbered from the right, 0 through 127, as shown in Figure 2–4.

Figure 2–4: H_floating Representation



The address of an H_floating-point value is the address of the byte containing bit 0 (address A in Figure 2–4). The form of an H_floating-point value is signed magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 0 are an excess 16,384 binary exponent.
- Bits 127 through 16 are a normalized 113-bit fraction, with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31.

The 15-bit exponent field encodes the values 0 through 32,767 as follows:

- An exponent value of 0 with a sign bit of 0 indicates that the H_floating-point number has a value of 0.
- Exponent values of 1 through 32,767 indicate binary exponents of –16,383 through +16,383.

If you are doing unchecked conversions to floating-point types (see Chapter 13 of the *VAX Ada Language Reference Manual*), note that an exponent value of 0 with a sign bit of 1 is considered to be a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

In VAX Ada, the VAX H_floating representation is used to represent the set of model numbers shown in Table 2–3 and the set of safe numbers shown in Table 2–4. On VAX machines, the value of an H_floating-point number is in the approximate range of $0.84*10^{-4932}$ through $0.59*10^{4932}$. The precision of an H_floating-point value is approximately one part in 2^{112}, or 33 decimal digits.

2.1.4 Fixed-Point Types and Objects

Fixed-point types provide approximations to the real numbers, with absolute bounds on errors determined by the value T'SMALL, where T is the fixed-point type. T'SMALL is defined to be less than or equal to the delta specified in the type declaration. In the absence of a length representation clause for T'SMALL, the delta value determines the value of T'SMALL, and the model numbers chosen for the type are determined from the value of T'SMALL and the specified range.

VAX Ada supports only values of T'SMALL that are powers of two between 2.0^{-62} and 2.0^{31}, inclusive. The VAX Ada compiler chooses the largest possible value of T'SMALL that is not greater than the specified delta, regardless of the range.

Values for objects of a fixed-point type are represented in VAX Ada as signed or unsigned, two's complement (binary) numbers multiplied by the value of T'SMALL. You can use length representation clauses to achieve unsigned representations; see Section 2.2.2.

In VAX Ada, the storage size for an object of any fixed-point type is determined by its delta and range and rounded up to an 8-, 16-, or 32-bit boundary. You can change the size with a representation clause (see Section 2.2 of this manual and Chapter 13 of the *VAX Ada Language Reference Manual*).

Operations on fixed-point types truncate the result towards 0.0, unless the language specifies otherwise.

Note that both model numbers and model intervals are used to define the permissible legal values for the results of operations on real (in this case, fixed-point) types. Any value that falls in the defined model interval for an operation is a legal result value for that operation. Thus, when you are working with fixed-point numbers, you may obtain results that are not what you expect in some cases. (See Chapter 4 of the *VAX Ada Language Reference Manual* for more information on model intervals and operations involving real types.)

For example, consider the following declaration:

```
type FP_TYPE is delta 0.1 range 0.0..1.0;
```

Because there is no representation clause for the type FP_TYPE, FP_TYPE'SMALL is 0.0625 (2^{-4}); 0.0625 is the largest power of 2 that is not greater than the delta (0.1). Now, suppose that your program uses an object of type FP_TYPE as follows:

```
A: FP_TYPE := 0.1;
 . . .
A := 3*A;
```

Because FP_TYPE'SMALL is 0.0625, and the model numbers used to represent objects of the type FP_TYPE are multiples of 0.0625, the model numbers for A are 0.0625, 0.125, 0.1875, 0.25, and so on up to 1.0. In this case, the model interval for A is 0.0625..0.125; the model interval for 3*A is 3*0.0625..3*0.125, or 0.1875..0.375.

Because 0.125 is too large, it is not a possible value for A. However, the lower bound, 0.0625, is a possible, and legal, value for A. For reasons of efficiency and to guarantee that the value of 3*A is also legal, the compiler could choose 0.0625 for A. Then, 3*A would result in 0.1875, which may be rounded up when printed out with an input-output procedure (rounding occurs when the FORE or AFT parameters constrain the number of decimal digits that the input-output procedure can print).

If FP_TYPE'SMALL were instead 0.03125 (either because of a different delta or because of a representation clause), the model interval for A would be 0.09375..0.125. Again, 0.125 is too large, but this time if the lower bound, 0.09375, is chosen for the value of A, 3*A results in 0.28125. This value is closer to the expected value, and is rounded up to 0.3 when printed out.

Therefore, when working with fixed-point types, and the results are not what you expect, consider tuning the distance between the underlying model numbers by using a representation clause (see Chapter 13 of the *VAX Ada Language Reference Manual*).

2.1.5 Array Types and Objects

In VAX Ada, an object of an array type is stored as a vector of equally spaced storage cells, one cell for each component. Any space between the components is assumed to belong to the array object, and may or may not be read or written by operations on the object. Thus, the storage size for an object of an array type is determined by the following equation:

$$number\ of\ components * (component\ size + distance\ between\ components)$$

Multidimensional arrays are stored in row-major order, and the components of all VAX Ada arrays are byte-aligned by default.

To force bit alignment and/or to minimize gaps, you must use the pragma PACK with the array type declaration.

For example, consider the following declarations:

```
type COLORS is (RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET);
type SPECTRUM is array(1..10) of COLORS;
WHITE_LIGHT: SPECTRUM;
```

Here, because values of the type COLORS are stored in a byte (see Section 2.1.1), and SPECTRUM has 10 components of the type COLORS, 10 bytes are allocated for the object WHITE_LIGHT.

In the next example, the object CHAR_ARRAY is stored in 30 bytes (thirty 8-bit components):

```
subtype INT is INTEGER range 1..10;
type TWO_DIM_ARRAY is
   array (INT range <>, INT range <>) of CHARACTER;
CHAR_ARRAY: TWO_DIM_ARRAY(1..5,5..10);
```

2.1.6 Record Types and Objects

In VAX Ada, the representation chosen for objects of a record type depends on a complex interaction among any applicable representation clauses and the types and subtypes of the record components. VAX Ada does not place any implementation-defined components within the object. For example, if the offset from the start of the object to a particular component depends on a value of a discriminant of the object, that offset is recalculated rather than stored in a "hidden" component in the record. This implementation allows you to explicitly specify all of the components of a record object, and to expect the result to be suitable for mixed-language programming.

Record objects are laid out so that all components affected by record representation clauses are first placed at the specified storage places; the remaining components are then laid out in the order in which they appear in the record declaration, discriminants first. Variants are overlaid and any alignment requirements of the components are met.

Thus, in the following example, the components are laid out in the order I, J, A, and B:

```
type SIMPLE_ARRAY is array (INTEGER range <>) of BOOLEAN;
type SIMPLE_LAYOUT (I,J: INTEGER) is
   record
      A: INTEGER;
      B: SIMPLE_ARRAY(I..J);
   end record;
```

Consider another example:

```
type SHOW_LAYOUT (DISCRIMINANT: BOOLEAN) is
   record
      A: INTEGER;
      case DISCRIMINANT is
         when TRUE  => B: CHARACTER;
         when FALSE => C: INTEGER;
      end case;
   end record;
```

Here, the components are laid out so that DISCRIMINANT appears first, then A. Then, because they are not affected by representation clauses, the variants are laid out starting on the first byte boundary after A.

If the type SHOW_LAYOUT from the preceding example were declared with a representation clause that specifically placed a component of one of the variants elsewhere, then that variant would be laid out first. Thus, if SHOW_LAYOUT were declared with the following representation clause, the compiler would lay out B first, then DISCRIMINANT, then A, then C:

```
for SHOW_LAYOUT use
   record
      B at 0*8 range 0..7;
   end record;
```

When working with records with discriminants, be aware that the offset from the start of the record object to a particular component may depend on the values of the discriminants, and thus may differ from one object to another. Similarly, the sizes of record objects of the same type may vary because of different discriminant values.

Within any record type, components whose sizes cannot be determined until run time cause succeeding components unaffected by representation clauses to be allocated at run-time-computed offsets from the start of the record. A component whose size or position cannot be determined until run time is called a *dynamic component.*

The dynamic calculation of component offsets and sizes may be done when the type is elaborated, or it may be done later—when the subtypes of all of the components have been forced, when the type itself is forced, or even at the point where the component is selected (this happens when the actual value of a discriminant is needed to make the calculation).

Thus, in the following example, A and B are both dynamically allocated: A because it is a dynamic component (an array with variable bounds), and B because its offset depends on the size of A:

```
type COMPONENT_ARRAY is array (INTEGER range <>) of INTEGER;
type ANOTHER_ORDER (I,J: INTEGER) is
   record
      A: COMPONENT_ARRAY(I..J);
      B: INTEGER;
   end record;
```

The laying out of a record type allows the compiler to determine the size of the type, where the size of the type is also the size of the largest possible object of that type. The size is related not to the sum of the sizes of the record's components, but to where the last component was laid out, including any allowances that were made for alignments. In other words, the size of a record type is computed as the position of the last component that physically appears in the layout plus the size of the last component (rounded up to a byte boundary if necessary). (Rounding depends on whether or not the record type itself is packable; see Section 2.2.1.)

Consider the following example:

```
type BIT_ARRAY is
   array (INTEGER range <>, INTEGER range <>) of BOOLEAN;
pragma PACK (BIT_ARRAY);
subtype N is INTEGER range 1..25;
type OFFICE_SECTION_LAYOUT (LENGTH : N := 1;
                            WIDTH  : N := 1) is
   record
      OCCUPIED : BIT_ARRAY(1..LENGTH, 1..WIDTH);
   end record;
FLOOR1 : OFFICE_SECTION_LAYOUT;
```

Here, the component OCCUPIED is an array of 1-bit components whose bounds depend on the values of LENGTH and WIDTH. When an unconstrained object, such as FLOOR1, is declared, it must be allocated enough storage to accommodate a value in which LENGTH and WIDTH could have any value in the range 1..25. For example, FLOOR1 could be assigned the following aggregate:

```
FLOOR1 := (20, 25, (1..20 => (1..25 => FALSE)));
```

Because the storage size allocated for an object like FLOOR1 must be adequate for any value that could be assigned to that object, the storage size must be the maximum storage size for the object. (The maximum storage size for an object is equal to the size of the type of the object.)

For example, you can calculate the maximum storage size of FLOOR1 as follows. The maximum values for LENGTH and WIDTH are each 25, and the largest possible OCCUPIED component is a 25-by-25 array (625 1-bit components). Because LENGTH and WIDTH are each of an integer subtype, one longword (32 bits) is allocated for each; 625 bits are allocated for the component OCCUPIED. The type is not packable (it does not have a compile-time constant size of 32 or fewer bits; see Section 2.2.1), so the estimated storage is rounded up to a byte boundary. Therefore, a total of 88 bytes ((32 + 32 + 625 + rounding bits)/8) will be allocated for FLOOR1.

The exact calculation of the size of a record can be nontrivial. For example, the size of the following record type can be calculated only by determining each possible record object and then choosing the largest result (which occurs when the value of the discriminant D is 5 or 6):

```
subtype INT is INTEGER range 1..10;
type TWO_DIM_ARRAY is
   array (INT range <>, INT range <>) of CHARACTER;
type REC (D: INT := 1) is
   record
      A: TWO_DIM_ARRAY(1..D,D..10);
   end record;
REC_OBJECT: REC;
```

The compiler uses simplifying assumptions to calculate the size of the type REC (REC'SIZE is also the maximum storage size for the object REC_OBJECT). These assumptions can cause the size allocated (or the values returned by the SIZE and MACHINE_SIZE attributes) to be different from what you might otherwise expect.

For example, if you manually calculate the number of bits required for component A, and add that to the number of bits required for discriminant D, you will arrive at one answer. Alternatively, if you ask the compiler for REC_OBJECT'SIZE (or REC_OBJECT'MACHINE_SIZE, as described in Section 2.2.7), you will receive a different answer. In fact, the compiler's

answer is based on a value of 10 for the upper bound of the first dimension, and a value of 1 for the lower bound of the second dimension. Therefore, the assumed maximum number of elements is 100, and the assumed storage size—(100*8)+32—is 832 bits.

See Section 2.2.7 of this manual and Chapter 13 of the *VAX Ada Language Reference Manual* for more information on the size attributes.

2.1.7 Access Types and Objects

VAX Ada uses a VAX virtual address to represent the value of an access type; the storage size for this value is a longword (32 bits). The objects designated by values of an access type are sized and represented according to their specified types. If the designated type is an unconstrained array, the virtual address points to an array descriptor that is chosen by the same rules used for choosing descriptors during parameter passing (see Chapter 5).

NOTE

These addresses do not necessarily point directly to objects of the access type. Thus, it is unsafe (as well as nonportable) to use the predefined generic procedure UNCHECKED_CONVERSION to convert between addresses and access types. Unchecked conversion between VAX addresses and access types is safe *only* when the accessed object is of a record type.

Each nonderived access type is associated with a collection, which is storage to be used for the objects designated by the type when allocators of that type are evaluated. If you specify a nonzero value in a length representation clause for the access type, that value determines the number of bytes (rounded up to an integral number of 512-byte pages) to be allocated for the collection associated with the type. The collection is not extended if it is exhausted. If you specify a zero value (or no length representation clause), the effective size of the collection is all of the available memory; no initial allocation is made, and the collection is extended as needed.

The collection associated with an access type is released when the scope of the access type is exited. See Section 2.3.2 for more information on storage deallocation. See Chapter 13 of the *VAX Ada Language Reference Manual* for more information on length representation clauses and collections. VAX Ada does not do automatic garbage collection.

In the following example, a 512-byte (1-page) collection is initially allocated for the access type NUM_PTR. One allocator is evaluated for FIRST_NUM, and 64 allocators are evaluated in the loop. Each evaluation causes 8 bytes of storage to be allocated as follows:

- The designated object in each case is of the type NUM_RECORD, and thus requires a longword (32 bits, or 4 bytes) for the integer component NUM.
- Each access type component (NEXT_NUM) requires a longword (32 bits, or 4 bytes).

When I reaches 63, a total of 64 allocators has been evaluated, and the collection limit has been reached. When I reaches 64, the collection limit is exceeded and not extended, and the exception STORAGE_ERROR is raised.

```
-- Procedure to construct a linked-list of integers.
--
procedure COLLECTION is

   type NUM_RECORD;
   type NUM_PTR is access NUM_RECORD;
   for NUM_PTR'STORAGE_SIZE use 512;
   type NUM_RECORD is
      record
         NUM: INTEGER;
         NEXT_NUM: NUM_PTR;
      end record;

   FIRST_NUM,ASSIGN_NUM: NUM_PTR;
   I: INTEGER;

begin

   FIRST_NUM := new NUM_RECORD'(0,null);
   ASSIGN_NUM := FIRST_NUM;
   for I in 1..64 loop
      ASSIGN_NUM.NEXT_NUM := new NUM_RECORD'(I,null);
      ASSIGN_NUM := ASSIGN_NUM.NEXT_NUM;
   end loop;

end COLLECTION;
```

2.1.8 Address Types and Objects

VAX Ada uses a VAX virtual address to represent the value of an object of the type SYSTEM.ADDRESS. The storage size for an object of an address type is a longword (32 bits).

2.1.9 Task Types and Objects

VAX Ada uses a VAX virtual address to represent the value of an object of a task type. The storage size for an object of a task type is a longword (32 bits).

When you declare an object of a task type, the value of the object is used by the VAX Ada run-time library to determine the address of the task control block created for the task. See Chapter 8 for more information on task storage allocation.

2.2 Data Optimization

VAX Ada provides type representation clauses, address clauses, and the language-defined pragma PACK to allow you to tailor the representation of nonpredefined types. Type representation clauses also allow you to control the representation of any new or derived types that you declare.

2.2.1 Pragma PACK

The predefined pragma PACK allows you to minimize gaps between the components of composite types (record and array types). When you apply the pragma PACK to a VAX Ada record or array type declaration, it has an effect only on the record or array components that are packable. In VAX Ada, a component is packable if its type allows it to be aligned on an arbitrary bit boundary.

For example, if you use the pragma PACK to pack an array of BOOLEAN components, any gaps between the components are minimized because enumeration type components are packable. However, the pragma PACK has no effect on an array of floating-point components.

Table 2–5 lists the type categories provided in VAX Ada and shows whether or not components of each type are packable.

Table 2–5: Packable Types

Type Category	Considered Packable as a Type	Affected by the Pragma PACK if a Component of a Record or Array
Integer	Yes	Yes
Enumeration[1]	Yes	Yes
Fixed-point	Yes	Yes
Floating-point	No	No
Address	No	No
Access	No	No
Task	No	No
Record	Depends[2]	Only if packable
Array	Depends[3]	Only if packable

[1]The predefined enumeration type CHARACTER (in the package STANDARD) is implemented as though the following declaration occurred in the package STANDARD: **for** CHARACTER'SIZE **use** 8. Thus, even in the presence of the pragma PACK, composite-type components of the type CHARACTER (or derived from the type CHARACTER) are not packed into 7 bits.

[2]Only if the record type has a compile-time constant size that is less than or equal to 32 bits, and if all of its components are packable.

[3]Only if the array type is itself a packed array of packable arrays, or if it is an array of 1-bit components. Components of the predefined array type STRING are not packable because the type STRING does not have 1-bit or packable array components.

Table 2–6 shows the effect of the pragma PACK on arrays and records with packable components.

Table 2–6: Effects of Packing the Components of Arrays and Records

	With Length Representation Clause on Component Type	Without Length Representation Clause on Component Type
With the Pragma PACK	Space between array and record components is minimized. Component size is determined by the length clause.	Space between array and record components is minimized. Component size is the default allocation for the component type.
	Saves only as much space as the length clause allows.	Saves the maximum amount of storage space.
Without the Pragma PACK	Space between array and record components is not minimized. Component size is determined by the length clause.	Space between array and record components is not minimized. Component size is the default allocation for the component type.
	Saves only as much space as the length clause and the default byte alignment allow.	Saves no storage space.

In the following example, the pragma PACK is used to minimize gaps in an array of fixed-point numbers:

```
type SMALL_FIXED_POINT is
   delta 2.0**(-4) range 0.0..0.5;
type SMALL_FIXED_POINT_ARRAY is
   array (INTEGER range <>) of SMALL_FIXED_POINT;
pragma PACK (SMALL_FIXED_POINT_ARRAY);
```

If SMALL_FIXED_POINT_ARRAY were not packed, the space-saving benefit of the small range of the SMALL_FIXED_POINT components would be lost. All of the components would be aligned on byte boundaries, causing 8-bit instead of the expected 3-bit component representations, and increasing the array size.

The next example shows the difference in space saving when length representation clauses are involved (see Section 2.2.2 for more information on length clauses):

```
type SMALL_INTEGER is new INTEGER range 0..7;
for SMALL_INTEGER'SIZE use 4;

type UNPACKED_SMALL_INTEGER_ARRAY is array (1..10) of SMALL_INTEGER;

type PACKED_SMALL_INTEGER_ARRAY is array (1..10) of SMALL_INTEGER;
pragma PACK (PACKED_SMALL_INTEGER_ARRAY);
```

In this example, the range of type SMALL_INTEGER causes it to require only 3 bits; however, the length clause specifies a size of 4 bits. For the array UNPACKED_SMALL_INTEGER_ARRAY, the length clause is honored for the SMALL_INTEGER components. However, because the array is declared without the pragma PACK, all of the components will be aligned on byte boundaries, and each component will have an effective size of 8 bits, instead of 4; the size of the array will be 80 bits. For the array PACKED_SMALL_INTEGER_ARRAY, each component will have a size of 4 bits, and any extra space between the components is eliminated; the size of the array will be 40 bits.

When using the pragma PACK, you must be careful to pack at the appropriate level: the pragma packs the components with respect to each other; it does not pack the subcomponents of the components closer together. In the following example, the size of the record UNPACKED_COMPONENTS is significantly larger than the size of the record PACKED_COMPONENTS, even though both are declared with the pragma PACK:

```
type UNSIGNED_INTEGER is new INTEGER range 0..7;
for UNSIGNED_INTEGER'SIZE use 3;

type PACKED_ARRAY is array (1..10) of BOOLEAN;
pragma PACK (PACKED_ARRAY);

type UNPACKED_ARRAY is array (1..10) of BOOLEAN;

type UNPACKED_COMPONENTS is
   record
      A,B: UNSIGNED_INTEGER;
      C:   UNPACKED_ARRAY;
   end record;
pragma PACK (UNPACKED_COMPONENTS);

type PACKED_COMPONENTS is
   record
      D,E: UNSIGNED_INTEGER;
      F:   PACKED_ARRAY;
   end record;
pragma PACK (PACKED_COMPONENTS);

BIG_RECORD:     UNPACKED_COMPONENTS; -- Size is 88 bits.
COMPACT_RECORD: PACKED_COMPONENTS;   -- Size is 16 bits.
```

Note that the pragma PACK never forces a component that begins a record variant off of a byte boundary. Such components are allocated on the next byte boundary. To force a component that begins a record variant to a boundary other than a byte boundary, you must use a record representation clause (see Sections 2.1.6 and 2.2.4 of this manual and Chapter 13 of the *VAX Ada Language Reference Manual*).

2.2.2 Length Representation Clauses

Length representation clauses allow you to explicitly control the amount of storage allocated for objects of a particular type.

The following example shows how length representation clauses are useful for declaring unsigned integer and unsigned fixed-point objects:

```
type UNSIGNED_INTEGER is new INTEGER range 0..2**16-1;
for UNSIGNED_INTEGER'SIZE use 16;

type UNSIGNED_FIXED_POINT is
   delta 2.0**(-8) range 0.0..255.0*2**(-8);
for UNSIGNED_FIXED_POINT'SIZE use 8;
```

The first declaration causes objects of the type UNSIGNED_INTEGER to be stored as unsigned words, rather than as signed longwords. The second declaration causes objects of the type UNSIGNED_FIXED_POINT to be stored as unsigned bytes, rather than as signed words. Note that because of Ada language rules, arithmetic operations involving these objects is signed. See Chapter 10 for more information on working with unsigned numbers.

A length representation clause is also useful for controlling the size of components in packed arrays. For example:

```
type SMALL_INTEGER is new INTEGER range 0..7;
for SMALL_INTEGER'SIZE use 4;

type SMALL_INTEGER_ARRAY is array (1..16) of SMALL_INTEGER;
pragma PACK (SMALL_INTEGER_ARRAY);
```

In this example, the range of SMALL_INTEGER and the use of the pragma PACK would cause the size of each component of SMALL_INTEGER_ARRAY to be 3 bits. However, the length representation clause causes the size of each component in the packed array to be 4 bits.

The *VAX Ada Language Reference Manual* gives complete information on the use of length representation clauses. Table 2–6 gives information on the interaction between length representation clauses and the pragma PACK.

2.2.3 Enumeration Representation Clauses

Enumeration representation clauses allow you to specify the internal codes that represent the literals of an enumeration type. When you use an enumeration representation clause, the storage size of each enumeration type is the amount of storage required to represent the full range of codes specified.

For example:

```
type ANSWER is (YES, NO, UNDECIDED);
for ANSWER use (YES => 0, NO => 8, UNDECIDED => 65535);
MY_UNSIGNED_ANSWER: ANSWER;
```

Here, the storage allocated for MY_UNSIGNED_ANSWER is a word. Even though only three integer codes must be represented, a word (16 bits) is needed to store values in the range 0..65535.

If any of the internal codes specified by the representation clause are negative, the representation for the type is signed; otherwise, it is unsigned. Thus, if you were to redeclare the type ANSWER as follows, the internal codes would be signed:

```
type ANSWER is (NO, YES, UNDECIDED);
for ANSWER use (NO => -8, YES=> 0, UNDECIDED => 65535);
MY_SIGNED_ANSWER: ANSWER;
```

Note that the signed representation requires an additional sign bit. To meet both the range of values (0..65535) and the signed representation, the storage allocated for MY_SIGNED_ANSWER is a longword.

2.2.4 Record Representation Clauses

Record representation clauses allow you to force a record type to have a particular representation. Thus, they are useful anytime you need to lay out an area of memory in a particular way. For example, you can use a record with a record representation clause to lay out a series of objects in a particular order. Or, you can use record representation clauses to lay out record types that declare objects that may be passed to other-language routines (including VMS system routines and VMS Run-Time Library routines). In particular, it is good programming practice to specify the layout of any record that is read from or written to an external file.

The following example defines a 16-bit mask for specifying the protection of a file (this definition is taken from the VAX Ada predefined package STARLET, which also defines the interfaces for various system service and VMS RMS routines). The mask contains four 4-bit fields, each of which specifies the protection to be applied to file access attempts by one of the four categories of user (system, owner, group, world).

```
-- Record defining a 4-bit field, each bit
-- representing a level of file protection.
--
type FILE_PROTECTION_FLAGS_TYPE is
   record
      NOREAD  : BOOLEAN;
      NOWRITE : BOOLEAN;
      NOEXE   : BOOLEAN;
      NODEL   : BOOLEAN;
   end record;

for FILE_PROTECTION_FLAGS_TYPE'SIZE use 4;
for FILE_PROTECTION_FLAGS_TYPE use
   record
      NOREAD  at 0 range 0..0;
      NOWRITE at 0 range 1..1;
      NOEXE   at 0 range 2..2;
      NODEL   at 0 range 3..3;
   end record;

pragma PACK (FILE_PROTECTION_FLAGS_TYPE);

-- Record defining a 16-bit mask that determines
-- the kind of file protection for each kind of
-- user; the record representation clause lays
-- out the 4-bit fields so that they are contiguous
-- half-bytes.
--
type FILE_PROTECTION_REC_TYPE is
   record
      SYS : FILE_PROTECTION_FLAGS_TYPE;
      OWN : FILE_PROTECTION_FLAGS_TYPE;
      GRP : FILE_PROTECTION_FLAGS_TYPE;
      WLD : FILE_PROTECTION_FLAGS_TYPE;
   end record;

for FILE_PROTECTION_REC_TYPE use
   record
      SYS at 0 range 0..3;
      OWN at 0 range 4..7;
      GRP at 1 range 0..3;
      WLD at 1 range 4..7;
   end record;

for FILE_PROTECTION_REC_TYPE'SIZE use 16;
```

When declaring record types with variants, you can use record representation clauses to conserve space. For example:

```
package ALIGN_VAR is

   type SMALL_INT is new INTEGER range 0..7;
   for SMALL_INT'SIZE use 3;

   type VARIANT_RECORD (VAR: BOOLEAN) is
      record
         A: SMALL_INT;
         case VAR is
            when TRUE  => X: CHARACTER;
                          Y: SMALL_INT;
            when FALSE => Z: SMALL_INT;
         end case;
      end record;

   for VARIANT_RECORD use
      record
         A   at 0 range 0..2;
         VAR at 0 range 3..3;
         X   at 0 range 4..11;
         Y   at 0 range 12..14;
         Z   at 0 range 4..6;
      end record;

end ALIGN_VAR;
```

Here, the representation clause on the type VARIANT_RECORD causes the variant, VAR, to be aligned on a bit boundary; when an object is declared and a case choice is made, the appropriate component is stored starting on bit 4 of the word of the storage allocated for the record object. (Without the representation clause, the variants would be aligned on byte boundaries.)

If you declare a record type with fixed-size components that follow (or are interspersed with) varying-size components, you will generate slower, less efficient code than if you declare a record type where all of the fixed-size components precede the varying-size components. For example:

```
package SLOW_LAYOUT is

   type VARYING_ARRAY is array (INTEGER range <>) of BOOLEAN;

   type SLOW_RECORD (I,J: INTEGER) is
      record
         A: INTEGER;
         B: VARYING_ARRAY(I..J);
         C: INTEGER;
         D: VARYING_ARRAY(I..I);
   end record;
   SLOW_OBJECT: SLOW_RECORD(1,10);

end SLOW_LAYOUT;
```

Here, the layout for the type SLOW_RECORD can be set up by the compiler only to the point of SLOW_RECORD.B; the rest of the layout and the allocation of storage for SLOW_OBJECT must be done at run time. Furthermore, each time you access SLOW_OBJECT.B, the size of SLOW_OBJECT.A must be calculated, thus decreasing the efficiency of any code that uses SLOW_OBJECT.

If the logical layout of a record type such as SLOW_RECORD is important, you can improve the efficiency of your code by declaring the type with a representation clause that forces the fixed-size components to known locations. For example:

```
package NOT_SO_SLOW_LAYOUT is

   type VARYING_ARRAY is array (INTEGER range <>) of BOOLEAN;
   pragma PACK (VARYING_ARRAY);

   type FASTER_RECORD(I,J: INTEGER) is
      record
         A: INTEGER;
         B: VARYING_ARRAY(I..J);
         C: INTEGER;
         D: VARYING_ARRAY(I..I);
      end record;

   for FASTER_RECORD use
      record
         I at  0 range 0..31;
         J at  4 range 0..31;
         A at  8 range 0..31;
         C at 12 range 0..31;
      end record;

   FASTER_OBJECT: FASTER_RECORD(1,10);

end NOT_SO_SLOW_LAYOUT;
```

Here, FASTER_OBJECT will be laid out so that the components fall in the following order: I, J, A, C, B, and D. The type representation clause forces the allocation of the components FASTER_OBJECT.B and FASTER_OBJECT.D to the end of the record.

Note that when you use a record representation clause to request a very small storage space for a component of a nonfixed-point discrete type, the record component value may be biased (its value may be predictably altered). When biasing occurs, the value stored is the unsigned quantity formed by subtracting COMPONENT_SUBTYPE'FIRST from the original value. (Because subtraction or addition is required to assign or fetch from the

component storage location, the generated code uses slightly more machine time than the unbiased form.)

Thus, in the following example, the values of R.C will be biased to allow them to be stored in the 2 bits required by the record representation clause (without the record representation clause, they would be stored as longwords):

```
subtype S is INTEGER range 100..103;
type R is
   record
      C : S;
   end record;
for R use
   record
      C at 0 range 0..1;
   end record;
 . . .
O : R;
 . . .
O.C := 100;
 . . .
```

2.2.5 Alignment Clauses

When you use a record representation clause to define the layout of a particular record type, you have the option of specifying an alignment clause to determine the alignment of all record objects (including record objects that are components) of that type. The *VAX Ada Language Reference Manual* gives the syntax and rules for using alignment clauses.

In VAX Ada, records can be aligned on any byte address that is a power of 2, up to 512 (or 2^9). Thus, in the following fragment, the value of ALIGN_AT could be any integer in the series 1, 2, 4, 8, . . . , 512. A value of 1 indicates byte alignment, a value of 2 indicates word alignment, and a value of 512 indicates page alignment.

```
type SMALLNUM is new INTEGER range 0..7;
for SMALLNUM'SIZE use 3;

ALIGN_AT: constant := 2;

type ALIGNED_RECORD is
   record
      A1: BOOLEAN;
      A2: SMALLNUM;
   end record;
```

```
for ALIGNED_RECORD use
   record at mod ALIGN_AT;
      A1 at 0 range 0 .. 0;
      A2 at 0 range 1 .. 3;
   end record;

type SHOW_ALIGNMENT is
   record
      S1,S2,S3: ALIGNED_RECORD;
   end record;
```

In this example, the components of the record SHOW_ALIGNMENT are aligned on 2-byte (word) boundaries, and the record SHOW_ALIGNMENT itself is aligned so that its component alignment can be observed. If the value of ALIGN_AT were 16, then the components of the record SHOW_ALIGNMENT would be aligned on 16-byte boundaries.

If you were to declare an array of components of the type ALIGNED_RECORD, and apply the pragma PACK to the array (which would be legal because the components of ALIGNED_RECORD are packable, and the record could have a compile-time size of less than 32 bits), the pragma would have no effect because the alignment clause interacts with the pragma.

VAX Ada places some restrictions on the possible alignments, depending on how objects of an aligned type are allocated (see Chapter 13 of the *VAX Ada Language Reference Manual* for a list of the restrictions). For example, a record object declared in a subprogram will be stack allocated, and thus can be aligned only **at mod** 1, 2, or 4 (it can be only byte-, word-, or longword-aligned). To force another alignment, you would have to declare the record type and object in a library package (it would then be statically allocated, and there would be no restrictions). Alternatively, you could declare an access type that designates the record; the designated object would be dynamically allocated, and, again, there would be no restrictions. See Section 2.3 for more information on dynamic storage allocation.

Alignment clauses can be useful in a mixed-language environment, where you may want to force objects to particular boundaries. Note, however, that the VAX hardware generally requires very little alignment; only a few instructions and VMS Run-Time Library routines need alignments (for example, queue and interlocked instructions). VAX Ada currently generates very few interlocked instructions.

2.2.6 Address Clauses

In VAX Ada, address clauses allow you to store objects (constants and variables) at specific memory locations. Thus, you can use address clauses to precisely map memory areas and overlay memory areas during program execution. Chapter 13 of the *VAX Ada Language Reference Manual* gives the syntax and rules for using address clauses; in particular, note the following rules or effects:

- A program that uses address clauses to overlay two or more Ada objects is erroneous.
- When you declare an object with an address clause, the usual implicit or explicit initialization associated with the type of the object is performed. Thus, access values are initialized to **null**, and record components may also receive initial values.

When you declare an object without an address clause, the compiler chooses an appropriate location for storing the object. However, when you specify an address clause, the compiler does not check that the address you have specified is appropriate. Thus, when you use address clauses, you need to be sure that you choose values that are meaningful in the VMS environment.

One way to obtain a meaningful value is to use the VMS Run-Time Library routine LIB$GET_VM to obtain a storage location. Example 2–1 is a complete procedure showing the use of an address clause to overlay an Ada record object onto storage allocated by LIB$GET_VM. The *VMS RTL Library (LIB$) Manual* describes LIB$GET_VM in more detail. For general information on the VMS environment (including information on VMS virtual address space), see the *VAX Hardware Handbook*.

Example 2–1: Using an Address Clause and LIB$GET_VM

```
with CONDITION_HANDLING; use CONDITION_HANDLING;
with SYSTEM; use SYSTEM;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with TEXT_IO; use TEXT_IO;
with LIB;
procedure USE_ADDRESS_CLAUSE is

   -- Declare a record for which storage will be allocated
   -- by the VMS Run-Time Library routine LIB$GET_VM; and
   -- freed by LIB$FREE_VM.
   --
   subtype STRING_14 is STRING(1..14);
   type OBJ_REC is
      record
         A: CHARACTER;
         B: INTEGER;
         C: STRING_14;
      end record;

   -- Declare the values needed to be passed to LIB$GET_VM and
   -- LIB$FREE_VM.
   --
   NUM_BYTES: constant INTEGER := OBJ_REC'MACHINE_SIZE/8;
   BASE_ADDR: ADDRESS;
   STATUS: COND_VALUE_TYPE;

begin

   -- Allocate the storage for a record of type OBJ_REC.
   --
   LIB.GET_VM (STATUS, NUM_BYTES, BASE_ADDR);
   if not CONDITION_HANDLING.SUCCESS(STATUS)
      then
         PUT_LINE("Failed to allocate memory");
      else
         PUT("Address of allocated storage is ");
         PUT(TO_INTEGER(BASE_ADDR));
         NEW_LINE;
   end if;

   -- Declare an object of type OBJ_REC, and place it at the
   -- storage location obtained with LIB$GET_VM using an
   -- address clause.
   --
   declare
      OBJECT: OBJ_REC;
      for OBJECT use at BASE_ADDR;
      O: STRING_14 := "Time for fun..";
```

(continued on next page)

Example 2–1 (Cont.): Using an Address Clause and LIB$GET_VM

```
   -- Do some useful work with the record object, and
   -- then free the storage by calling LIB$FREE_VM.
   --
   begin
      OBJECT := (A => 'A', B => 5, C => "Summer is a...");
      PUT_LINE(OBJECT.C);
      OBJECT.C := O;
      PUT_LINE(OBJECT.C);
   end;
   LIB.FREE_VM (STATUS, NUM_BYTES, BASE_ADDR);

end USE_ADDRESS_CLAUSE;
```

2.2.7 Determining the Sizes of Types and Objects

VAX Ada provides a number of methods for determining how much storage has been allocated for particular types and objects:

- You can use the predefined attributes T'SIZE and T'MACHINE_SIZE to determine the number of bits used and allocated for a given type or object.
- You can use the /WARNINGS=COMPILATION_NOTES qualifier on any of the compilation commands (DCL ADA and ACS COMPILE and RECOMPILE) to determine how record types and so on have been laid out.
- You can use the debugger (after compiling and linking your program) to examine the sizes of your variables.

The first of these methods is discussed in this section; the other two are described in *Developing Ada Programs on VMS Systems*.

As indicated by its name, the predefined SIZE attribute returns information on the size of a type or an object (see Chapter 13 of the *VAX Ada Language Reference Manual*). When using this attribute, note the following differences in the values it returns:

- T'SIZE (where T represents a type) returns the *minimum* number of bits needed to represent an object of the type.
- O'SIZE (where O represents an object) returns the *actual* number of bits used for the object's current value.

The minimum number of bits and the actual number of bits can often be quite different. For example, given the following declaration, the value of BOOL17'SIZE will be 1:

```
type BOOL17 is new BOOLEAN;
```

One bit is the minimum amount of storage required for an object of the type BOOL17. Even if you used a representation clause to declare BOOL17, as in the following declaration, the value of BOOL17'SIZE will still be 1; when applied to types, the SIZE attribute is not affected by representation clauses:

```
type BOOL17 is new BOOLEAN;
for BOOL17'SIZE use 17;
```

The VAX Ada attribute T'MACHINE_SIZE provides similar information for a type or subtype that O'SIZE provides for an object. Table 2–7 summarizes the differences between T'SIZE, O'SIZE, and T'MACHINE_SIZE.

Table 2–7: Comparison of SIZE and MACHINE_SIZE Attribute Results

Type or Subtype	T'SIZE	O'SIZE	T'MACHINE_SIZE
Discrete or fixed-point without length clause	Minimum number of bits needed to represent an object of the type or subtype T.	Actual number of bits used by O. If O is not a record or array component or is unpacked, the result is the same as the T'MACHINE_SIZE result for O's subtype. If O is a packed component, the result is the number of bits needed so that components can be packed as tightly as possible.	Total number of bits allocated for an object of the subtype. Result is the actual number of bits used, rounded up to an 8-, 16-, or 32-bit boundary; representation is signed.

(continued on next page)

Table 2–7 (Cont.): Comparison of SIZE and MACHINE_SIZE Attribute Results

Type or Subtype	T'SIZE	O'SIZE	T'MACHINE_SIZE
Discrete or fixed-point with length clause	Minimum number of bits needed to represent an object of the type or subtype T.	Actual number of bits used by O; length clause determines upper bound (except if O is a component of a record specified with a component clause).	Total number of bits allocated for an object of the type or subtype T. Result is the actual number of bits used, rounded up to an 8-, 16-, or 32-bit boundary; representation can be unsigned.
All other types, with or without length clauses	Minimum number of bits needed to represent an object of the type or subtype T.	Actual number of bits used by O.	Total number of bits allocated for an object of the type T. Result is the actual number of bits used, rounded up to a byte boundary.

Note that the T'MACHINE_SIZE of a base type can be equal to or greater than the T'SIZE of the same base type. The T'MACHINE_SIZE of a base type can also be less than the T'SIZE of a first named subtype that has an associated size representation clause. Consider the following declarations:

```
type INT8 is range 0..255;
for INT8'SIZE use 8;
I: INT8;
```

An examination of INT8 and I produces the following results:

INT8'SIZE	8
INT8'MACHINE_SIZE	8
I'SIZE	8
INT8'BASE'SIZE	16
INT8'BASE'MACHINE_SIZE	16

The number of bits needed to represent the specified range values symmetrically about 0 is 16, so that INT8'BASE'SIZE is 16. This value is greater than the values of INT8'MACHINE_SIZE, INT8'SIZE, and I'SIZE. Note that the values of INT8'MACHINE_SIZE and I'SIZE are equal, as they should be.

Table 2–8 gives a set of results for a variety of interesting cases.

Table 2–8: Results of Size Attributes for Various Types and Objects

Declaration and Attributes	Number of Bits
type BOOL17 **is new** BOOLEAN; **for** BOOL17'SIZE **use** 17; B: BOOL17;	
Type BOOL17'SIZE	1
Object B'SIZE	17
Type BOOL17'MACHINE_SIZE	32
Type BOOL17'BASE'SIZE	1
Type BOOL17'BASE'MACHINE_SIZE	32
type ET **is range** 0..255; **for** ET'SIZE **use** 8; E: ET;	
Type ET'SIZE	8
Object E'SIZE	8
Type ET'MACHINE_SIZE	8
Type ET'BASE'SIZE	16
Type ET'BASE'MACHINE_SIZE	16
type NET **is new** ET **range** 0..7; NE: NET;	
Type NET'SIZE	3
Object NE'SIZE	8
Type NET'MACHINE_SIZE	8
Type NET'BASE'SIZE	16
Type NET'BASE'MACHINE_SIZE	16
type NT **is new** INTEGER **range** 0..255; **for** NT'SIZE **use** 8; N:NT;	

(continued on next page)

Table 2–8 (Cont.): Results of Size Attributes for Various Types and Objects

Declaration and Attributes	Number of Bits
Type NT'SIZE	8
Object N'SIZE	8
Type NT'MACHINE_SIZE	8
Type NT'BASE'SIZE	32
Type NT'BASE'MACHINE_SIZE	32
C: CHARACTER;	
Type CHARACTER'SIZE	7
Object C'SIZE	8
Type CHARACTER'MACHINE_SIZE	8
Type CHARACTER'BASE'SIZE	7
Type CHARACTER'BASE'MACHINE_SIZE	8
type BIT_ARRAY **is array** (1..10) **of** BOOLEAN; **pragma** PACK (BIT_ARRAY); BA: BIT_ARRAY;	
Type BIT_ARRAY'SIZE	10
Object BA'SIZE	10
Type BIT_ARRAY'MACHINE_SIZE	16
Type BIT_ARRAY'BASE'SIZE	10
Type BIT_ARRAY'BASE'MACHINE_SIZE	16

2.3 Storage Allocation and Deallocation

To make efficient use of storage from your VAX Ada programs, you need to know how and where objects are stored. You also need to know how and when objects, particularly objects designated by access types, are deallocated. The following sections give information on both of these topics.

2.3.1 Storage Allocation

The VAX Ada compiler stores objects in registers, on a stack, in static memory, or in dynamic memory (on the heap) depending upon the objects' size, when their size is known, their type, how long their lifetimes are, and how they are used.

If you take the address of an object (O'ADDRESS), an implicit pragma VOLATILE is assumed for the object within the scope of the subprogram or task where the address is taken. Within that scope, the object is guaranteed to be allocated at a unique memory location, regardless of where the object is declared. If the object is also declared within that scope, the object is allocated in memory for the duration of the object's lifetime; the object receives a unique memory address, and keeps that address from the time it is elaborated until the time when its containing scope is left. See Chapter 10 for more information on working with address values. See the *VAX Ada Language Reference Manual* and Chapter 8 for more information on the pragma VOLATILE.

If you have specified an object with an IMPORT_OBJECT, EXPORT_OBJECT, or PSECT_OBJECT pragma, the object is initialized each time it is elaborated. See the *VAX Ada Language Reference Manual* and Chapter 5 for more information on these pragmas.

The compiler always stores objects created by allocators in dynamic memory. In accordance with Ada language rules, the dynamic memory allocated for each access type is structured as a *collection*. A collection is a memory area that comes into existence when the access type is elaborated, and goes out of existence when the scope containing the access type is left. Each time an allocator is evaluated, storage for the resulting object is allocated from the collection belonging to the corresponding access type. There is some CPU overhead involved both when the collection is allocated and when the collection is deallocated (see Section 2.3.2 for more information on storage deallocation).

By default, no storage is initially allocated for a collection; storage is allocated as needed, until all virtual memory is depleted. You can change the default behavior with a length clause (see the *VAX Ada Language Reference Manual*). See Section 2.1.7 for more information on the representation and allocation of objects of access types.

You may be able to improve the efficiency of your program by carefully sizing the collections allocated for access types. When you use a length representation clause (T'STORAGE_SIZE) to specify the sizes of access type collections, choose values that will be integrally related after they have been rounded up to a number of pages (T'STORAGE_SIZE specifies the number

of bytes to be used for a collection; in VAX Ada, this number is rounded up to an integral number of 512-byte pages). For example, values of 512*4, 512*8, and 512*12 are better than values of 512*2, 512*7, and 512*13. There is no common denominator for 2, 7, and 13, but there is a common denominator for 4, 8, and 12.

This practice will result in reduced fragmentation of memory; also, when you free several collections (implicitly) at scope exit, the freed storage will most likely be in blocks large enough to be useful for other collections.

2.3.2 Storage Deallocation

VAX Ada does not provide garbage collection. However, there are at least two ways in which you can deallocate objects of access types:

- Make use of the fact that the collection associated with an access type is automatically deallocated when the end statement of the scope containing the access type is encountered.
- Instantiate the language-defined generic procedure UNCHECKED_DEALLOCATION and call the instantiation to explicitly deallocate the storage for an object designated by a value of an access type (see Chapter 13 of the *VAX Ada Language Reference Manual* for the syntax of UNCHECKED_DEALLOCATION).

  When you call an instantiation of UNCHECKED_DEALLOCATION, storage is deallocated for the object within the collection allocated for the access type. Thus, the effect of using UNCHECKED_DEALLOCATION is to conserve the use of the collection, rather than to deallocate the collection for general use by your program.

Note that the collections for access types declared in library packages are not deallocated until the entire program has completed executing. The only way you can conserve the use of such storage is to use an instantiation of the procedure UNCHECKED_DEALLOCATION.

Example 2–2 shows a main program that depends on an access type declared in a library package. The program uses an instantiation of the procedure UNCHECKED_DEALLOCATION to deallocate the storage for the access type.

Example 2–2: Using UNCHECKED_DEALLOCATION to Control Access Type Storage Deallocation

```
-- Package containing declarations of access type and
-- corresponding deallocation procedure. Collection size is
-- set using a length clause, to simulate a limited-storage
-- application.
--
with UNCHECKED_DEALLOCATION;
package ACCESS_TYPES is

   type LIST_ELEMENT_CLASS is (HEAD,ELEMENT);
   type LIST_ELEMENT(CLASS: LIST_ELEMENT_CLASS);
   type LIST_ELEMENT_PTR is access LIST_ELEMENT;
   for LIST_ELEMENT_PTR'STORAGE_SIZE use 4*512;
   type LIST_ELEMENT (CLASS: LIST_ELEMENT_CLASS) is
      record
         NEXT: LIST_ELEMENT_PTR;
         case CLASS is
            when ELEMENT => ELEMENT_VALUE: INTEGER;
            when HEAD    => HEAD_VALUE: INTEGER := 0;
         end case;
      end record;

   procedure FREE_ELEMENT is
      new UNCHECKED_DEALLOCATION(LIST_ELEMENT,
                                 LIST_ELEMENT_PTR);

end ACCESS_TYPES;

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

-- Main program that demonstrates how a collection can be used up
-- quickly: the main program creates a 65-element linked list
-- (including the header); the block inside the program creates an
-- array of tasks, which, in turn, create linked lists of various
-- lengths. If the access types used by the tasks were declared
-- only in the block, the storage would be deallocated at the end
-- of the block. Because the types are declared in a library
-- package used by both the main program and the block, the
-- collection for the access type is maintained until the main
-- program finishes and exits. Unchecked deallocation must be
-- used instead to conserve use of collection storage.
--
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with ACCESS_TYPES; use ACCESS_TYPES;
procedure CONTROL_STORAGE is
```

(continued on next page)

Example 2–2 (Cont.): Using UNCHECKED_DEALLOCATION to Control Access Type Storage Deallocation

```
   -- Procedure to create and initialize a unidirectional linked
   -- list of integers; the parameter to the procedure determines
   -- the list length.
   --
   procedure MAKE_LIST (Y : in INTEGER) is

      HEAD_ELEMENT: LIST_ELEMENT_PTR := new LIST_ELEMENT(HEAD);
      THIS_ELEMENT, NEXT_ELEMENT: LIST_ELEMENT_PTR;
      N : INTEGER := Y;

   begin

      -- Create and initialize values of list, starting at the
      -- first element.
      --
      THIS_ELEMENT := HEAD_ELEMENT;
      for I in 1..N loop
         THIS_ELEMENT.NEXT := new LIST_ELEMENT'(CLASS => ELEMENT,
                                                NEXT  => null,
                                                ELEMENT_VALUE => I);
         THIS_ELEMENT := THIS_ELEMENT.NEXT;
      end loop;

      -- Do something with the linked list...and then deallocate
      -- the storage.
      --
      loop
         THIS_ELEMENT := HEAD_ELEMENT.NEXT;
         exit when THIS_ELEMENT = null;
         HEAD_ELEMENT.NEXT := THIS_ELEMENT.NEXT;
         FREE_ELEMENT(THIS_ELEMENT);
      end loop;

   end MAKE_LIST;

begin

   -- Create (and deallocate) the list for the main program.
   --
   MAKE_LIST(64);
```

(continued on next page)

Example 2–2 (Cont.): Using UNCHECKED_DEALLOCATION to Control Access Type Storage Deallocation

```
   -- Concurrently, create (and deallocate) a series of
   -- varied-length lists used by an array of tasks.
   --
   INNER_BLOCK:
       declare
          task type USE_SPACE is
             entry NUM_ELEMENTS (X : in INTEGER);
          end USE_SPACE;

          type TASK_ARRAY is array (1..10) of USE_SPACE;
          SPACE_ARRAY: TASK_ARRAY;

          task body USE_SPACE is
          begin
             accept NUM_ELEMENTS (X : in INTEGER) do
                MAKE_LIST(X);
             end;
          end USE_SPACE;

       begin
          for I in SPACE_ARRAY'RANGE loop
             SPACE_ARRAY(I).NUM_ELEMENTS(I);
          end loop;
       end INNER_BLOCK;

end CONTROL_STORAGE;
```

Chapter 3

Input-Output Facilities

Although VAX Ada allows you to invoke VMS input-output system services and VMS Record Management Services (RMS) directly (see Chapters 5 and 6), for most applications it is not necessary to do so. The VAX Ada predefined input-output packages provide a rich and comprehensive set of file operations, and each input-output package is tailored for operations on a specific kind of file.

VAX Ada predefines the following packages:

SEQUENTIAL_IO
DIRECT_IO
RELATIVE_IO
INDEXED_IO
SEQUENTIAL_MIXED_IO
DIRECT_MIXED_IO
RELATIVE_MIXED_IO
INDEXED_MIXED_IO
TEXT_IO

Of these, the packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO are predefined by the Ada language; the rest are predefined by the VAX Ada implementation. All of the package specifications, as well as explanations of the operations provided by each package, are presented in Chapter 14 of the *VAX Ada Language Reference Manual*.

The VAX Ada predefined packages and their operations are implemented using VMS RMS file organizations and facilities. This chapter describes the implementation and explores some of its implications.

The information in this chapter is based on the information in Chapter 14 of the *VAX Ada Language Reference Manual*. You should also be familiar with VMS RMS file organizations and access methods, know how to work with VMS file specifications and directories, and be familiar with the VMS File Definition Language (FDL).

If you need introductory information on VMS file specifications and directories or FDL, see the *Guide to VMS File Applications*. For more information about VMS RMS and VMS RMS services, see the *VMS Record Management Services Manual*; for more information on FDL, see the *VMS File Definition Language Facility Manual*.

3.1 Files and File Access

To input and output data to and from an Ada program, you must first associate the file objects in your program with external files. All of the VAX Ada input-output packages supply CREATE and OPEN procedures that allow you to make this association:

- Each CREATE procedure creates a new external file and then associates a file object with it.
- Each OPEN procedure associates a file object with an existing external file.

Thus, in the following example, EXTERNAL_FILE.TXT is created only once, but it is associated with both file objects ONE_FILE and ANOTHER_FILE at different points in the procedure:

```
with TEXT_IO; use TEXT_IO;
procedure MAKE_FILES is
   ONE_FILE:     FILE_TYPE;
   ANOTHER_FILE: FILE_TYPE;
begin
   -- Create EXTERNAL_FILE.TXT and associate it with
   -- the file object ONE_FILE.
   --
   CREATE (FILE => ONE_FILE,
           NAME => "EXTERNAL_FILE.TXT");
    . . .
   -- Close EXTERNAL_FILE.TXT and disassociate it with
   -- the file object ONE_FILE.
   --
   CLOSE  (ONE_FILE);
    . . .
   -- Reopen EXTERNAL_FILE.TXT and associate it with
   -- a different file object.
   --
   OPEN   (FILE => ANOTHER_FILE,
           MODE => OUT_FILE,
           NAME => "EXTERNAL_FILE.TXT");
    . . .
end MAKE_FILES;
```

When you create or open a VAX Ada file object, the external file with which it is associated is a VMS RMS file that has a particular kind of organization and that allows a particular kind of access. Each element in the file is associated with a VMS RMS record that has a particular kind of format. A default organization, access, and record format is determined by the input-output package that you use to create the file. Depending on the package, you can change these defaults with a CREATE or OPEN FORM parameter.

Section 3.3 discusses the FORM parameter and system-dependent external file attributes in more detail; Sections 3.6.1 through 3.6.4 and 3.7 provide tables of default attributes for each VAX Ada input-output package.

The following sections summarize how file objects, called Ada files in this chapter, and external files (VMS RMS files) are related. See Chapter 14 of the *VAX Ada Language Reference Manual* for detailed definitions of Ada files; see the *Guide to VMS File Applications* for detailed definitions of VMS RMS file organizations and record formats.

3.1.1 Ada Sequential Files

An Ada *sequential file* is a set of file elements occupying consecutive positions in linear order. Values are transferred in the order in which they are read or written to the file, and when you open a file, the transfer starts from the beginning of the file.

You can associate an Ada sequential file with a VMS RMS file of any organization; the records in the VMS RMS file can have fixed-length, variable-length, variable-length with fixed-length control (VFC), or stream format.

The packages SEQUENTIAL_IO and SEQUENTIAL_MIXED_IO provide sequential access to Ada sequential files.

3.1.2 Ada Direct Files

An Ada *direct file* is a set of file elements occupying consecutive positions in linear order. You can transfer values to or from an element of the file at any selected position. The position of an element is specified by its index, which is an integer in the range from 1 to $2^{31} - 1$ (a value of the subtype POSITIVE_COUNT). The first element, if any, has an index of 1; the index of the last element, if any, is called the current size. The current size is zero if there are no elements.

An open Ada direct file has a current index, which is set to 1 when you create or open the file. The current index determines which element will be involved in the next read or write operation.

You can associate an Ada direct file only with a VMS RMS file with sequential organization; the records in the VMS RMS file must have fixed-length format.

The packages DIRECT_IO and DIRECT_MIXED_IO provide direct access to Ada direct files.

3.1.3 Ada Relative Files

An Ada *relative file* is a set of fixed-length cells occupying consecutive positions in linear order. Cells in a relative file are numbered from 1 to $2^{31} - 1$ (the numbers are values of the subtype POSITIVE_COUNT); the number of a cell is called its index. The cells in a relative file can either be empty or can contain fixed- or variable-length elements.

An open Ada relative file has a current index, which is set to 1 when the file is created or opened. The current index determines which element will be involved in the next read or write operation. The concept of size does not apply to relative files; end-of-file is true if, starting at the current index, all cells are empty.

You can associate an Ada relative file only with a VMS RMS file with relative organization; the records in the VMS RMS file can have fixed-length, variable-length, or variable-length with fixed-length control (VFC) format.

The packages RELATIVE_IO and RELATIVE_MIXED_IO provide relative access to Ada relative files.

3.1.4 Ada Indexed Files

An Ada *indexed file* is a set of file elements that are ordered by predefined keys. Each element has at least one primary key (numbered 0), and may have as many as 254 alternate keys (numbered 1 to 254). You define keys in the form string (in the FORM parameter) when the file is created. The elements of an indexed file can be accessed by any key.

An open Ada indexed file has a next element, which is the first element determined by the primary key when the file is first opened; the next element is redefined after each successful read operation, or it may be reset to the first sequential element according to the specified key. The concept of size does not apply to Ada indexed files: end-of-file is true if, starting at next element in the file, no elements exist.

You can associate an Ada indexed file only with a VMS RMS file with indexed organization; the records in the VMS RMS file can have fixed-length or variable-length format.

The packages INDEXED_IO and INDEXED_MIXED_IO provide indexed access to Ada indexed files.

3.1.5 Ada Text Files

An Ada *text file* is a sequence of pages, where a page is a sequence of lines, and a line is a sequence of characters. Characters, lines, and pages are all numbered starting from 1 and range to $2^{31} - 1$ (the numbers are values of the subtype POSITIVE_COUNT). The number of a character is called its column number. The line terminator that marks the end of a line has a column number that is one more than the number of characters in the line.

The current column number in a text file is the column number of the next character or line terminator to be read or written. Similarly, the current line number is the number of the current line, and the current page number is the number of the current page.

You can associate an Ada text file only with a VMS RMS file with sequential organization. The records in the VMS RMS file can have fixed-length, variable-length, or variable-length with fixed-length control (VFC) format.

The package TEXT_IO provides sequential access to Ada text files.

3.2 Naming External Files

In VAX Ada, you identify external files using VMS file specifications. All of the VAX Ada input-output packages have CREATE and OPEN procedures which, in turn, have a NAME parameter that allows you to associate the name of an external file with a particular file object. The NAME parameter can have one of two values:

- A string that denotes a VMS file specification or a logical name. If the value of NAME is a file specification, the Ada file object given by the FILE parameter in the particular CREATE or OPEN procedure is associated with an external file named by that specification.
- A null string (the default). If the value of NAME is a null string, then the external file is a temporary file that is deleted when the file is closed. Temporary files have no file name; however, they are created using the file specification SYS$SCRATCH:. To redirect temporary files to another device, redefine the logical name SYS$SCRATCH to name a different device. Note that because temporary files are not entered in a directory, they cannot inherit the file ownership of any directory.

The CREATE and OPEN procedures also have a FORM parameter that allows you to identify an external file (see Section 3.3). In VAX Ada, the FORM parameter takes as its value a VMS FDL string or a reference to a file of FDL statements. By specifying a value for the FDL FILE DEFAULT_NAME attribute in a CREATE or OPEN FORM parameter, you can give file specification information that will be used by default if any of that information is omitted from the string given for the NAME parameter. Thus, in the following example, the external file will have the specification SOME_FILE.DAT:

```
CREATE(FILE => F,
       MODE => OUT_FILE,
       NAME => "SOME_FILE",
       FORM => "FILE; DEFAULT_NAME '.DAT'");
```

The value of the NAME parameter governs, even if you give a value using the FORM parameter and FDL attributes. For example, if you omit a value for the NAME parameter and try to specify a complete file name with the FDL FILE DEFAULT_NAME attribute, the default name is ignored, and the external file is still a temporary file that is deleted when the file is closed.

You cannot use the FDL FILE NAME attribute to name an external file; a value specified with that attribute is ignored.

The following sections summarize how to write and use logical names in place of file specifications. For a full description of file specifications and logical names, see the *VMS DCL Concepts Manual* and the *Guide to VMS File Applications*.

3.2.1 File Specification Syntax

A *file specification* identifies an external file or a device on the VMS operating system. The syntax is as follows:

```
node::device:[directory]filename.type;version
```

node
Is the name of a network node. This element applies only to systems that are part of a network (systems that support DECnet-VAX).

device
Is the name of the physical device on which the file is stored or is to be written. The device name is the only part of a file specification that is used for record-oriented devices (such as printers and card readers).

directory
Is the name of the directory (and any subdirectories) under which the file is cataloged on the specified device. You must delimit the directory name with square brackets ([]), as shown in the syntax description, or with angle brackets (<>). You must use a period to separate a series of directories or subdirectories within the square or angle brackets. Directory names apply only to files stored on disk devices (as opposed to files stored on tape).

filename
Is the name of the file; the maximum length is 39 characters. The allowed characters are upper- or lowercase letters, digits, underscore (_), hyphen (-), or dollar sign ($). A file name specification is appropriate only for files stored on mass storage devices (such as disks and tape).

type
Is the type of the file; the maximum length is 39 characters. The allowed characters are upper- or lowercase letters, digits, underscore (_), hyphen (-), or dollar sign ($). The type must begin with a letter or digit. By convention, the type is an abbreviation that describes the kind of data in the file. You must use a period to separate the file name and type. A type specification is appropriate only for files stored on mass storage devices.

version
Is a decimal number that specifies which version of the file is desired. The version number is incremented by one each time a new version of a file is created. The maximum version number is 32767. You can refer to version numbers in a relative manner by specifying 0 as the latest (highest numbered) version of the file, –1 as the next most recent version, –2 as the version before that, and so on. You can use either a semicolon, as shown in the syntax description, or a period to separate type and version. A version number is appropriate only for files stored on mass storage devices (such as disks and tape).

The maximum size of a file specification, including all delimiters, is 255 characters.

You do not need to explicitly state all of the elements of a file specification. If you omit an element, a default value is applied. For more information, see the *VMS DCL Concepts Manual*.

You can use VAX Ada form strings (that is, the value of the FORM parameter in an input-output package CREATE or OPEN procedure) to further define or change default file specifications. See Section 3.3.3.

3.2.2 Logical Names

A *logical name* is a name that represents a file, directory, or physical device. Every logical name is paired with an equivalence string (or list of equivalence strings). An *equivalence string* is a character string denoting a full file specification, a device name, or another logical name. Thus, logical names are a convenient shorthand for file names to which you refer frequently. See the *VMS DCL Concepts Manual* and *Guide to VMS File*

Applications for complete explanations of logical names and examples of their use. See also the descriptions of the DCL ASSIGN and DEFINE commands in the *VMS DCL Dictionary* or *VMS General User's Manual.*

Logical names are maintained by the system in four logical name tables: your process table, the job table for your process, your group table, and the system table. These tables are described in the *VMS DCL Concepts Manual.*

By default, VMS creates a set of logical names for you when you log in. Table 3–1 lists the predefined names that are most relevant to VAX Ada input-output.

Table 3–1: Predefined (Default) Logical Names

Logical Name	Table in Which the Name is Stored	What the Name Represents
SYS$COMMAND	Process	Original (first-level) SYS$INPUT stream.
SYS$DISK	Process	Default device established at login or changed by the DCL SET DEFAULT command.
SYS$ERROR	Process	Default device or file to which the system writes error messages generated by warnings, errors, and severe errors.
SYS$INPUT	Process	Default input stream for the process.
SYS$LOGIN	Job	Device and directory established at login time as the home directory for the process.

(continued on next page)

Table 3–1 (Cont.): Predefined (Default) Logical Names

Logical Name	Table in Which the Name is Stored	What the Name Represents
SYS$NET	Process	The source process that invokes a target process in DECnet-VAX task-to-task communication. When opened by the target process, SYS$NET represents the logical link over which that process can exchange data with its partner. SYS$NET is defined only during task-to-task communication. (Task-to-task communication refers to tasks that are VMS images running in the context of a process, not Ada tasks.)
SYS$OUTPUT	Process	Default output stream for the process.
SYS$SCRATCH	Job	Default device and directory to which temporary files are written.
TT	Process	Default device name for terminals.
ADA$INPUT	Determined by user	Default device or file from which Ada TEXT_IO input is read; SYS$INPUT if not defined by the user.
ADA$OUTPUT	Determined by user	Default device or file to which Ada TEXT_IO output is written; SYS$OUTPUT if not defined by the user.

The names SYS$COMMAND, SYS$ERROR, SYS$INPUT, and SYS$OUTPUT represent process-permanent files (files that are open for the life of your process). They have different equivalence strings associated with them depending on whether they are used interactively, in a batch job, or in a command procedure. You can also redefine them. The *VMS DCL Concepts Manual* explains and demonstrates the use of these names; Table 3–2 shows the source of the equivalence strings associated with them.

Note that you can achieve asynchronous input-output in tasking programs by defining the logical names ADA$INPUT and ADA$OUTPUT so that they refer to nonprocess-permanent files; for example, by defining ADA$INPUT and ADA$OUTPUT so that they refer to TT, you can achieve asynchronous terminal input-output. See Section 3.9.2 for more information.

Table 3–2: Equivalence Strings for Default Logical Names for Process-Permanent Files

Logical Name	Interactive Mode[1]	Batch Mode[1]	Command Procedure[1]
SYS$COMMAND	Terminal	Disk	Terminal
SYS$INPUT	Terminal	Disk	Disk
SYS$ERROR	Terminal	Log file	Terminal
SYS$OUTPUT	Terminal	Log file	Terminal

[1]Note the following definition of terms: *terminal* means the device name of your terminal; *disk* means the batch input or command file; and *log file* means the batch job log file.

3.3 Specifying External File Attributes

The CREATE and OPEN procedures in the VAX Ada input-output packages all have a FORM parameter that allows you to specify the system-dependent attributes of an external file. Most of the time you will not need to use the FORM parameter when you create or open a file because each input-output package assumes certain attributes for the external file by default (see Section 3.3.3). In fact, you never need to specify a value for FORM when you *open* an existing file. You do need to specify it under the following conditions when you *create* a file:

- With a relative or direct file where the item by which the input-output package is instantiated is unconstrained, you must specify the maximum size of the file elements (records) in bytes.
- With a relative mixed-type or direct mixed-type file, you must specify the maximum size of the file elements (records) in bytes.
- With an indexed file, you must specify information about the primary and any alternate keys.

The value of the FORM parameter must be a VMS FDL string, or it must be a reference to a file of FDL statements.

FDL is a special-purpose language that is written as an ordered sequence of file attribute keywords (sometimes called FDL statements) and their associated values. These keywords and values determine the characteristics of external files. By using an FDL string (or a reference to a file of FDL statements) as the value of the FORM parameter in a CREATE or OPEN input-output operation, you can give your file any of the VMS RMS

attributes available in FDL, and you thereby supersede the default file attributes of your input-output package (see Section 3.3.3).

If you are not familiar with FDL, see the *Guide to VMS File Applications*; it introduces FDL and shows how to design files using the Edit/FDL Utility. See the *VMS File Definition Language Facility Manual* for complete information about FDL, including specific definitions of the FDL statements. The following sections summarize the FDL concepts and statements that you need to know to specify file attributes in VAX Ada FORM parameters.

3.3.1 The VMS File Definition Language (FDL): Primary and Secondary Attributes

FDL statements—whether in an FDL file or in a VAX Ada form string—specify predefined VMS RMS file attributes. *Primary attributes* take a single value or represent a group of related, or *secondary*, attributes, which also take values. Most of the primary attributes that have secondary attributes do not themselves take a value. Table 3–3 lists the available primary and secondary attributes.

Table 3–3: FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
TITLE	Primary attribute gives a title to the FDL file.	None
IDENT	Primary attribute gives the date and time of creation of the FDL file, and specifies the name of the creating utility (either Edit/FDL or Analyze/RMS_File).	None
SYSTEM	Primary attribute takes no value. Secondary attributes give system identification information.	DEVICE, SOURCE, TARGET

(continued on next page)

Table 3–3 (Cont.): FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
FILE	Primary attribute takes no value. Secondary attributes determine file characteristics: its default name, owner, organization, protection, and revision; what will happen when it is opened or closed; whether or not data checking will be done when the file is read or written; what kind of processing is allowed; how much space is allocated for the file, and whether or not the space is contiguous; and so on. Secondary attributes also allow specification of magnetic tape file operations. Some FILE secondary attributes have corresponding AREA secondary attributes.	ALLOCATION, BEST_TRY_CONTIGUOUS, BUCKET_SIZE, CLUSTER_SIZE, CONTEXT, CONTIGUOUS, CREATE_IF, DEFAULT_NAME, DEFERRED_WRITE, DELETE_ON_CLOSE, DIRECTORY_ENTRY, EXTENSION, FILE_MONITORING, GLOBAL_BUFFER_COUNT, MAXIMIZE_VERSION, MAX_RECORD_NUMBER, MT_BLOCK_SIZE, MT_CLOSE_REWIND, MT_CURRENT_POSITION, MT_NOT_EOF, MT_OPEN_REWIND, MT_PROTECTION, NAME, NON_FILE_STRUCTURED, ORGANIZATION, OUTPUT_FILE_PARSE, OWNER, PRINT_ON_CLOSE, PROTECTION, READ_CHECK, REVISION, SEQUENTIAL_ONLY, SUBMIT_ON_CLOSE, SUPERSEDE, TEMPORARY, TRUNCATE_ON_CLOSE, USER_FILE_OPEN, WINDOW_SIZE, WRITECHECK

(continued on next page)

Table 3–3 (Cont.): FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
DATE	Primary attribute takes no value. Secondary attributes specify dates and times for backup, creation, expiration, and revision of the file. In general, the only secondary attribute that can be routinely and safely set is EXPIRATION; the others should be set by the system, and thus are not useful in an Ada FORM parameter.	BACKUP, CREATION, EXPIRATION, REVISION
RECORD	Primary attribute takes no value. Secondary attributes specify the characteristics of records in the file: their size; the kind of carriage control; and their format.	BLOCK_SPAN, CARRIAGE_CONTROL, CONTROL_FIELD, FORMAT, SIZE
ACCESS	Primary attribute takes no value. Secondary attributes specify the file-processing operations allowed on the file.	BLOCK_IO, DELETE, GET, PUT, RECORD_IO, TRUNCATE, UPDATE
NETWORK	Primary attribute takes no value. Secondary attributes set run-time network access parameters.	BLOCK_COUNT LINK_CACHE_ENABLE LINK_TIMEOUT NETWORK_DATA_CHECKING
SHARING	Primary attribute takes no value. Secondary attributes specify whether or not multiple readers or writers can concurrently access the file.	DELETE, GET, MULTISTREAM, PROHIBIT, PUT, UPDATE, USER_INTERLOCK

(continued on next page)

Table 3–3 (Cont.): FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
CONNECT	Primary attribute takes no value. Secondary attributes specify run-time attributes that are application dependent and related to record access and performance.	ASYNCHRONOUS, BLOCK_IO, BUCKET_IO, CONTEXT, END_OF_FILE, FAST_DELETE, FILL_BUCKETS, KEY_GREATER_EQUAL, KEY_GREATER_THAN, KEY_LIMIT, KEY_OF_REFERENCE, LOCATE_MODE, LOCK_ON_READ, LOCK_ON_WRITE, MANUAL_UNLOCKING, MULTIBLOCK_COUNT, MULTIBUFFER_COUNT, NOLOCK, NONEXISTENT_RECORD, READ_AHEAD, READ_REGARDLESS, TIMEOUT_ENABLE, TIMEOUT_PERIOD, TRUNCATE_ON_PUT, TT_CANCEL_CONTROL_O, TT_PROMPT, TT_PURGE_TYPE_AHEAD, TT_READ_NOECHO, TT_READ_NOFILTER, TT_UPCASE_INPUT, UPDATE_IF, WAIT_FOR_RECORD, WRITE_BEHIND

(continued on next page)

Table 3–3 (Cont.): FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
AREA	Primary attribute takes an integer value in the range 0 to 254, which identifies the area in an indexed file. (Multiple areas must have a separate AREA section defined for each.) Secondary attributes specify characteristics of the area: how much space is allocated; whether or not the space is contiguous; positioning of the area; the volume on which the area will reside, and so on. Most AREA secondary attributes have corresponding FILE secondary attributes.	ALLOCATION, BEST_TRY_CONTIGUOUS, BUCKET_SIZE, CONTIGUOUS, EXACT_POSITIONING, EXTENSION, POSITION, VOLUME
KEY	Primary attribute takes an integer value in the range 0 to 254, which gives the number of a key in an indexed file; the primary key number must be 0. Secondary attributes specify the characteristics of keys in the indexed file.	CHANGES, COLLATING_SEQUENCE, DATA_AREA,DATA_FILL, DATA_KEY_COMPRESSION, DATA_RECORD_COMPRESSION, DUPLICATES, INDEX_AREA, INDEX_COMPRESSION, INDEX_FILL, LENGTH, LEVEL1_INDEX_AREA, NAME, NULL_KEY, NULL_VALUE, POSITION, PROLOG, SEGn_LENGTH, SEGn_POSITION, TYPE
ANALYSIS_OF_AREA	Result of using Analyze/RMS_File Utility; will appear only in FDL files that describe indexed files. Neither primary nor secondary attributes are useful in an Ada FORM parameter.	RECLAIMED_SPACE

(continued on next page)

Table 3–3 (Cont.): FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
ANALYSIS_OF_KEY	Result of using Analyze/RMS_File Utility; will appear only in FDL files that describe indexed files. Neither primary nor secondary attributes are useful in an Ada FORM parameter.	DATA_FILL, DATA_KEY_COMPRESSION, DATA_RECORD_COMPRESSION, DATA_RECORD_COUNT, DATA_SPACE_OCCUPIED, DEPTH, DUPLICATES_PER_SIDR, INDEX_COMPRESSION, INDEX_FILL, INDEX_SPACE_OCCUPIED, LEVEL1_RECORD_COUNT, MEAN_DATA_LENGTH, MEAN_INDEX_LENGTH

When using FDL to specify the attributes of an Ada external file, observe the following FDL rules. Any FDL errors occurring in a FORM parameter will raise the Ada predefined exception USE_ERROR.

- The primary attributes must appear in the order shown in Table 3–3.
- Each attribute string (primary or secondary) constitutes an FDL statement, and must be terminated with a semicolon. In the following example, RECORD, FORMAT FIXED, and SIZE 120 are three separate FDL statements:

```
-- Create SOME_FILE.DAT with fixed record format and
-- a record size of 120 bytes.
--
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT",
       FORM => "RECORD; FORMAT FIXED; SIZE 120;");
```

The exclamation point is the comment character in FDL, and anything following it is ignored. For example:

```
-- Create SOME_FILE.DAT with 80-byte records.
--
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT",
       FORM => "RECORD; SIZE 80; !80-byte records");
```

- Each FDL statement can represent only one primary or secondary attribute and its associated value. Each statement can have no more than 132 characters (including blanks). To format your program without adding extra blanks to the form string, use the Ada catenation operator (&) to break up the form string into individual statement strings. Thus, you could rewrite the preceding example as follows:

```
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT ",
       FORM => "RECORD; "              &
                  "FORMAT FIXED; "     &
                  "SIZE 120;"          );
```

- If you are working with an indexed file that has two or more AREA primary attributes, they must follow one another in numerical order.
- If you are working with an indexed file that has two or more KEY primary attributes, they must follow one another in numerical order. In addition, any SEGn secondary attributes must follow one another in numerical order, and the SEGn numbers must be dense. In other words, if you use SEG3 to label a key segment, then segments SEG0, SEG1, and SEG2 must also exist.
- Keywords can be truncated to their shortest unique abbreviations, and strings must be enclosed either in a pair of apostrophes (' ') or a pair of double quotation marks (" "). Note that Ada based integers or integers with underscores are not legal FDL syntax.

In addition to allowing you to specify file attributes directly in a form string, VAX Ada also allows you to give a reference to an FDL file using a VMS file specification. The specification must be preceded by an at sign (@). For example:

```
-- Create SOME_FILE.DAT with specifications declared in
-- the FDL file FILE_ATTRIBUTES.FDL.
--
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT",
       FORM => "@FILE_ATTRIBUTES.FDL");
```

An advantage of being able to give a reference to an FDL file is that you can use the Edit/FDL Utility to construct the FDL file. The utility is designed to help you choose file attributes that will help optimize the efficiency of your program. In particular, the utility is helpful in tuning indexed files. For example, it can plot graphs to help you determine appropriate bucket sizes for specific indexed files. See the *Guide to VMS File Applications* for more information on the Edit/FDL Utility and file design.

Table 3–4 describes the primary and secondary FDL attributes that you will be most likely to use in a VAX Ada program, and gives their default values. For convenience, primary attributes are shown in boldface type; secondary attributes are shown in regular type and indented. The intent of the table is to provide a quick reference and to summarize information presented in the *VMS File Definition Language Facility Manual*; see that manual for details.

As shown in the table, the value assigned to an attribute can take one of the following forms:

Switch — A logical value, set to TRUE, YES, FALSE, or NO. TRUE (or YES) sets the attribute; FALSE (or NO) clears it. (You can also use the abbreviations T, Y, F, and N for TRUE, YES, FALSE, and NO.)

Keyword — An actual word that you must type (in either upper- or lowercase) after the attribute name. You can truncate a keyword to its shortest unique abbreviation.

Integer — A decimal integer (based integers or underscores are not allowed).

String — A character string (enclosed in either a pair of apostrophes or a pair of double quotation marks) that you must type after the attribute name. The null string is a valid string value. Note that to use double quotation marks in the same statement, you must write the form string following Ada conventions. For example:

```
CREATE(FILE => F,
       MODE => OUT_FILE,
       FORM => "FILE;"                                  &
                  "DEFAULT_NAME ""SOME_FILE.DAT"";"&
                  -- A pair of quotation marks
                  -- inside a string represents one
                  -- quotation mark.
               "RECORD;"                                &
                  "FORMAT FIXED;"                       &
                  "SIZE 100;"                           );
```

Alternatively, you can use apostrophes to make your code easier to read:

```
CREATE(FILE => F,
       MODE => OUT_FILE,
       FORM => "FILE;"                                  &
                  "DEFAULT_NAME 'SOME_FILE.DAT';"  &
                  "RECORD;"                             &
                     "FORMAT FIXED;"                    &
                     "SIZE 100;"                        );
```

Table 3–4: Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
TITLE	String of up to 132 characters, including the TITLE keyword. No default value.	Names the FDL file.
IDENT	String of up to 132 characters, including the IDENT keyword. Default value is the date, time of creation, name of creating utility if created with Edit/FDL or Analyze/RMS_File; otherwise, no default value.	Record identifying file information.
SYSTEM		
DEVICE	String. Default value is a null string.	Comment (names the disk model on which the file will reside).
FILE		
ALLOCATION	Integer in the range 0 to 4294967295. Default value is 0.	Sets the number of blocks that will be initially allocated for the file. If 0, the system will not preallocate space for the file.
BEST_TRY_ CONTIGUOUS	Switch. Default value is NO.	Controls whether the file will be allocated contiguously if there is enough space for it. If set to YES, and there is enough space for the file, the file will be allocated contiguously; if there is not enough space, the file will not be allocated contiguously. If set to NO, this attribute is ignored.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
BUCKET_SIZE	Integer in the range 0 to 63. Default value is 0.	Sets the number of blocks per bucket. If 0, VMS RMS computes the bucket size to be the smallest bucket size capable of holding the largest record.
CONTIGUOUS	Switch. Default value is NO.	Controls whether the file must be allocated contiguously. When set to YES and there is not enough space for the file's initial allocation, an error message results. When set to NO or no allocation is specified, the attribute is ignored.
DEFAULT_NAME	String. Default value is a null string.	Uses its string value to define portions of the file specification of the file to be created. If you supply only a partial file specification in the NAME parameter to an Ada OPEN or CREATE operation, the DEFAULT_NAME value is used for the missing part of the file specification. If you have not specified a value for DEFAULT_NAME, the VMS RMS defaults are used for the missing part.
EXTENSION	Integer in the range 0 to 65535. Default value is 0.	Sets the number of blocks for the default extension value for the file. Each time the file is extended, the specified number of blocks is added. If 0, the extension size is determined by the system each time the file must be extended.
FILE_MONITORING	Switch. Default value is NO.	Turns on VMS RMS statistics gathering for subsequent use in doing performance analysis.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
MAX_RECORD_ NUMBER	Integer in the range 0 to 2147483647. Default value is 0.	Specifies the maximum number of records that can be placed in a relative file. If 0, then you can place as many records as you want in the file, up to 2,147,483,647 (or $2^{31} - 1$).
ORGANIZATION	Keyword. Default value is SEQUENTIAL.	Defines the kind of file organization. Value must be one of the keywords SEQUENTIAL, RELATIVE, or INDEXED.
PRINT_ON_CLOSE	Switch. Default value is NO.	Controls whether the data file is to be spooled to the process default print queue (SYS$PRINT) when the file is closed. When set to YES, the data file is spooled; when set to NO, the attribute is ignored. (This attribute applies to sequential files only.)
PROTECTION	String. Default value is the system or process default.	Defines the levels of file protection for the file. Its value can take one of two forms (SYSTEM=code, OWNER=code, GROUP=code, WORLD=code) or (SYSTEM:code, OWNER:code, GROUP:code, WORLD:code) where the code is a protection specification for READ, WRITE, EXECUTE, and DELETE in the form RWED. To deny a specific access right, you omit it from the code. To give no access rights to a user classification, you omit the classification from the list. For example, the following string gives all access rights to SYSTEM and OWNER, gives only READ access to GROUP, and gives no access rights to WORLD: (SYSTEM=RWED, OWNER=RWED, GROUP=R).

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
SEQUENTIAL_ONLY	Switch. Default value is NO.	Indicates that the file can only be processed sequentially, thus allowing certain processing optimizations. Any attempt to perform random access results in an error.
SUBMIT_ON_CLOSE	Switch. Default value is NO.	Determines whether the data file is submitted to the process batch queue (SYS$BATCH) when the file is closed. When set to YES, the data file is submitted to the process default batch queue; this setting makes sense only if the file is a command file with sequential organization. When set to NO, this attribute is ignored.
DATE		
EXPIRATION	String in the form dd-mmm-yyyy hh:mm:ss.cc. Default value is a null string.	Sets the date and time after which a disk file can be considered for deletion. For magnetic tape files, this attribute sets the date and time after which you can overwrite the file. This is the only DATE secondary attribute that you can routinely and safely set.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
RECORD		
CARRIAGE_ CONTROL	Keyword. Default value is CARRIAGE_RETURN.	Specifies the kind of carriage control for the records in the file. Value must be one of the keywords CARRIAGE_RETURN, FORTRAN, NONE, or PRINT. See Section 3.7.4 of this manual and the *VMS File Definition Language Facility Manual* for more information.
FORMAT	Keyword. Default value is VARIABLE.	Sets the record format for the data file. Value must be one of the keywords FIXED, STREAM, STREAM_ CR, STREAM_LF, UNDEFINED, VARIABLE, VFC. See the *VMS File Definition Language Facility Manual* for more information.
SIZE	Integer. No default value.	Sets the maximum record size in bytes. With fixed-length records, this value is the length of every record in the file. With variable-length records, this value is the length of the longest record that can be placed in the file. If the file has sequential or indexed organization, you can specify 0 and the system will not impose a maximum record length. The records in an indexed file, however, cannot cross bucket boundaries. If the file has relative organization, the SIZE attribute is used with the BUCKET_SIZE attribute to set the size of the fixed-length cells.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
		If the records have variable-length with fixed control (VFC) format, the fixed-control portion of the record is not included in the SIZE calculation; only the data portion is set by this attribute. The fixed area is the size, in bytes, of the fixed-control portion of VFC records. Regular variable-length records have a fixed-control size of 0. See the *VMS File Definition Language Facility Manual* for the maximum sizes allowed for the various record organizations and formats.
ACCESS		
DELETE	Switch. The default value is FALSE.	Permits VMS RMS delete operations.
GET	Switch. Default value is GET when a file is being opened and no other ACCESS secondary attribute has been specified and SHARING DELETE or SHARING UPDATE have been specified.	Permits VMS RMS get or find operations.
PUT	Switch. PUT when creating a file.	Permits VMS RMS put or extend operations.
TRUNCATE	Switch. Default value is FALSE.	Permits VMS RMS truncate operations.
UPDATE	Switch. Default value is FALSE.	Permits VMS RMS update or extend operations.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
SHARING		
DELETE	Switch. No default value.	Allows other users to delete records from the file.
GET	Switch. TRUE if ACCESS GET has also been specified.	Allows other users to read the file.
PROHIBIT	Switch. YES if ACCESS DELETE, ACCESS PUT, ACCESS TRUNCATE, or ACCESS UPDATE has been specified; otherwise, no default value.	Prohibits any kind of file sharing by other users. When set to YES, this attribute takes precedence over all other ACCESS secondary attributes. A value of YES in a VAX Ada form string takes precedence over any other default values that may be implied by values of other SHARING secondary attributes. When an OPEN or CREATE form string specifies any SHARING secondary attribute without specifying SHARING PROHIBIT, then no default is chosen (equivalent to a value of NO).
PUT	Switch. No default value.	Allows other users to write records to the file.
UPDATE	Switch. No default value.	Allows other users to update records that currently exist in the file.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
CONNECT		
MULTIBUFFER_ COUNT	Integer in the range 0 to 127. No default value.	Specifies the number of buffers to be allocated when the file is opened. If the value is not set or 0, VMS RMS chooses a default value (see the *VMS File Definition Language Facility Manual*). This attribute is ignored for DECnet operations.
READ_AHEAD	Switch. No default value.	Indicates read-ahead operations; to be used with multiple buffers. When one buffer is filled, the next record is read into the next buffer while the input-output operation takes place for the first buffer. Because the system does not have to wait for input-output completion, input and computing can overlap. This attribute is ignored for DECnet operations. See the *VMS File Definition Language Facility Manual* for more information.
TIMEOUT_ENABLE	Switch. No default value.	Specifies the maximum time, in seconds, that will be allowed for a record input wait (see TIMEOUT_ PERIOD). The input wait can be caused by a locked record if the WAIT_FOR_RECORD attribute has also been specified, or it can be caused by the input of a character from the terminal. If the timeout period expires, VMS RMS returns an error status. This attribute is ignored for DECnet operations.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
TIMEOUT_PERIOD	Integer in the range 0 to 255. No default value.	Specifies the maximum number of seconds that a VMS RMS get operation can take; if the operation is specified from the terminal and you specify 0, the current contents of the type-ahead buffer are returned. You must use the TIMEOUT_ENABLE attribute with TIMEOUT_PERIOD. This attribute is ignored for DECnet operations.
TRUNCATE_ON_PUT	Switch. No default value.	Specifies that a VMS RMS put or write operation can occur at any point in a file, truncating the file at that point. A write operation causes the end-of-file mark to immediately follow the last byte written. You can use this attribute only with VMS RMS sequential files.
UPDATE_IF	Switch. No default value.	Indicates that if a put operation is specified for a record that exists in the file, the operation is converted to an update. This attribute is necessary to overwrite (as opposed to update) an existing record in VMS RMS relative and indexed sequential files. Indexed files using this attribute must not allow duplicates on the primary key.
WAIT_FOR_RECORD	Switch. No default value.	Specifies that VMS RMS should wait for a currently locked record until it becomes available. You can use this attribute with the TIMEOUT_ENABLE and TIMEOUT_PERIOD attributes to limit waiting periods to a specified time.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
WRITE_BEHIND	Switch. No default value.	Indicates that write-behind operations are to occur when multiple buffers are used. When one buffer is filled, the next record is written into the next buffer while the input-output operation takes place for the first buffer. Because the system does not have to wait for input-output completion, computing and output can overlap. See the *VMS File Definition Language Facility Manual* for more information.
AREA	This attribute and its secondary attributes apply only to files with indexed organization. See the *VMS File Definition Language Facility Manual* for details.	
KEY	Integer in range 0 to 254. No default value.	Denotes the key number for a file with indexed organization. The value for the primary key must be 0; the value for alternate keys can be any integer in the range 1 to 254. This attribute and its secondary attributes apply only to files with indexed organization.
CHANGES	Switch. Default value is NO.	Determines whether or not key values can be changed with a VMS RMS update operation. Note that a value of YES for primary keys is an error; a value of YES for alternate keys is allowed.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
DATA_KEY_ COMPRESSION	Switch. Default value is YES.	Controls whether leading and trailing repeating characters in the primary key will be compressed. For compression to occur, you should define your indexed file as a Prolog 3 file with the FDL attributes KEY PROLOG; KEY PROLOG 3 is the default. You should set this attribute for indexed files involved in DECnet operations.
DATA_RECORD_ COMPRESSION	Switch. Default value is YES.	Controls whether repeating characters are compressed in data records. For compression to occur, your indexed file must be defined as a Prolog 3 file with the FDL attributes KEY PROLOG; KEY PROLOG 3 is the default. You should set this attribute for indexed files involved in DECnet operations.
DUPLICATES	Switch. Default value is NO for the primary key; YES for alternate keys.	Controls whether duplicate keys are allowed in files with indexed organization. When set to YES, this attribute specifies that there can be more than one record with the same specific key value. When set to NO, duplicate keys are not allowed, and any attempt to write a record where the key would be a duplicate will result in an error.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
INDEX_COMPRESSION	Switch. Default value is YES.	Controls whether leading repeating characters in the index are compressed. For compression to occur, you should define your indexed file as a Prolog 3 file with the FDL attributes KEY PROLOG; KEY PROLOG 3 is the default. You should set this attribute for indexed files involved in DECnet operations.
LENGTH	Integer. No default value.	Sets the length of the key, in bytes. This value, along with the POSITION and TYPE attributes, is used when the key is unsegmented. Because there is no default, this value must be specified.
NAME	String of from 1 to 32 characters. Default value is a null string.	Assigns a name to a key. This value is optional. The specified string is padded with ASCII null characters to a length of 32 bytes.
NULL_VALUE	Character or unsigned decimal integer representing an ASCII value. Default value is the ASCII null character (0).	Defines the null value that will instruct the system not to create an alternate index entry for the record that has the null value in every byte of the key field. If the alternate key is of the type STRING or DSTRING, you can specify the null value by either specifying the character itself or by specifying an unsigned decimal number denoting the character's ASCII value. To specify the character, enclose it in apostrophes; to specify the decimal ASCII value, type it without enclosing apostrophes.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
POSITION	Integer. No default value.	Defines the byte position of the beginning of the key field within the record. The first position is 0; primary keys work best if they start at byte 0. You can use this attribute along with the KEY LENGTH and TYPE attributes, when the key is unsegmented.
PROLOG	Integer in the range 1 to 3. Default value is the system or process default.	Defines the internal structure of a file with indexed organization. See the *VMS File Definition Language Facility Manual* for details.
SEGn_LENGTH	Integer in the range 0 to 7. No default value.	Defines the length of the key segment, in bytes. This attribute is used with the SEGn_POSITION attribute when the key is segmented. The "n" is the number of the segment and may be numbered from 0 to 7; the first segment must be numbered 0. Segmented keys must be of the type STRING or DSTRING, and segments may not overlap for Prolog 3 files.
SEGn_POSITION	Integer. No default value.	Defines the key segment's starting position within the record. The first position is 0. Segmented keys must be of the type STRING or DSTRING, and segments may not overlap for Prolog 3 files.

(continued on next page)

Table 3–4 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
TYPE	Keyword. Default value is STRING.	Defines the type of the key. May have any of the following values: BIN2, BIN4, BIN8, COLLATED, DCOLLATED, DBIN2, DBIN4, DBIN8, DECIMAL, DDECIMAL, DINT2, DINT4, DINT8, DSTRING, INT2, INT4, INT8, STRING. See the *VMS File Definition Language Facility Manual* for more information.

Certain FDL attributes can significantly improve application performance. In other words, if the files used by the application are designed and tuned properly, the application will run more efficiently, often because a minimum number of input-output operations will occur. File design and tuning are important for large files, especially indexed files. The characteristics you specify when you create a file often have a significant effect on application performance at run time.

The following FDL attributes from Table 3–4 can affect application performance:

FILE ALLOCATION
FILE BEST_TRY_CONTIGUOUS
FILE BUCKET_SIZE
FILE CONTIGUOUS
FILE EXTENSION
CONNECT READ_AHEAD
CONNECT WRITE_BEHIND
ACCESS and SHARING attributes
certain KEY attributes

The following attributes not listed in Table 3–4 can also affect performance:

FILE DEFERRED_WRITE
CONNECT FAST_DELETE
CONNECT GLOBAL_BUFFER_COUNT
CONNECT MULTIBLOCK_COUNT
CONNECT MULTIBUFFER_COUNT

FILE SEQUENTIAL_ONLY
FILE WINDOW_SIZE

See the *Guide to VMS File Applications* for more information.

3.3.2 Creation-Time and Run-Time Attributes

Of the many attributes that you can associate with an external file, some exist as long as the external file exists. These are called *creation-time attributes*. FILE ORGANIZATION and RECORD SIZE are examples of creation-time attributes.

The rest of the attributes exist only as long as the external file is associated with a particular file object. These are called *run-time attributes*. Any of the attributes secondary to the primary CONNECT, ACCESS, or SHARING attributes, as well as the FILE secondary PRINT_ON_CLOSE attribute, are run-time attributes. Run-time attributes can change dynamically at run time, and must be respecified each time the file is opened.

The *Guide to VMS File Applications* identifies all creation-time and run-time attributes and discusses them in more detail.

You can change a file's creation-time characteristics only by creating or recreating the file. Inside an Ada program, you can give creation-time attributes to an external file with a call to a CREATE procedure; the file inherits these attributes in subsequent calls to OPEN procedures. Outside of an Ada program, you can change the creation-time characteristics of an external file by using the Edit/FDL and Convert or Convert/Reclaim Utilities to create a new external file and populate it with elements of the old file.

Any creation-time file attributes specified in an OPEN procedure are considered to be only assertions; they do not affect the external file's characteristics. VAX Ada protects you from making wrong assertions of creation-time attributes in a call to an OPEN procedure. If you specify a form string value for the FORM parameter in an OPEN procedure call, the OPEN procedure checks the following creation-time attributes of the external file against any assertions of the attributes in the form string:

- The FILE secondary attribute ORGANIZATION
- The RECORD secondary attribute CARRIAGE_CONTROL
- The RECORD secondary attribute FORMAT
- The RECORD secondary attribute SIZE
- Every KEY section (in an indexed file)

If there is a mismatch, then the exception USE_ERROR is raised. For example, if a form string asserts that the organization of the external file is indexed, but the external file being opened is sequential, the exception USE_ERROR is raised. If no creation-time-attribute assertions are made, then no check is performed.

3.3.3 Default External File Attributes

When you open a file (using either a CREATE or an OPEN procedure), the input-output package you are using provides a set of default external file attributes. One purpose of the default attributes is to allow your program to pass a null form string (the default) to an OPEN procedure and still open the external file. Thus, you do not need a form string (a FORM parameter value) when you use an OPEN procedure to open a file. However, in some situations you must specify certain external file attributes when you call a CREATE procedure (see Section 3.3).

Sections 3.6.1 through 3.6.4 and Section 3.7 provide tables of default attributes for each VAX Ada input-output package. Note the following points about the default external file attributes:

- Creation-time attributes specified in the FORM parameter of an OPEN procedure have no effect, except to cause a consistency check against the creation-time attributes that exist for the file (see Section 3.3.2).
- Many FDL default attributes are applied automatically, but they are not shown in the default attribute tables; see the *VMS File Definition Language Facility Manual* for the FDL defaults. The VAX Ada input-output packages impose certain restrictions on the attributes of the external files that they open:
  - If the file is being created, these restrictions are checked against any external file characteristics given in the FORM parameter of the CREATE procedure.
  - If the file is being opened, the restrictions are checked after any assertions in the FORM parameter of the OPEN procedure have been checked against the existing attributes of the file.

  If the restrictions are violated at either point, the exception USE_ ERROR is raised.

3.4 File Sharing

File sharing in VAX Ada enables concurrent access to the same external file. In other words, file sharing permits multiple file objects to be associated with the same external file. File sharing can take place in the same VMS process or across multiple processes. You can share external files for reading, writing, or modification.

Because VAX Ada files are layered on VMS RMS file organizations, the rules that apply to read and write sharing of VMS RMS files also apply to Ada files. The *Guide to VMS File Applications* gives complete information on file sharing in the VMS environment. For descriptions of the organizations chosen for Ada files, see Section 3.1.

The FDL ACCESS and SHARING primary attributes have secondary attributes that control the scope of access and sharing of an external file. The ACCESS secondary attributes determine the kinds of operations (read, write, update, and so on) that your program can perform on the external file. The SHARING secondary attributes determine the kinds of operations other concurrently active programs can perform on the file.

When you open a file, VAX Ada uses the MODE parameter to select appropriate default ACCESS and SHARING secondary attributes (see Section 3.3.3 and Tables 3–5 through 3–13). If the FORM parameter in an OPEN or CREATE procedure specifies values for the ACCESS or SHARING attributes, those values supersede any previously specified or default values.

To determine whether or not you need to specify ACCESS or SHARING attributes, follow these steps:

1. Check the table of default attributes for the package you are working with. For example, if you are working with relative files, look at Table 3–9.
2. If the table does not show a default for a particular attribute, check Table 3–4 or the *VMS File Definition Language Facility Manual.*
3. If the combined set of default values does not reflect the action you want, use the form string to set the attribute values.

When choosing attribute values, note the following points:

- The ACCESS and SHARING attributes interact to some degree. For example, YES values for ACCESS DELETE, PUT, TRUNCATE, or UPDATE cause a value of YES for SHARING PROHIBIT.

- In any attempt to open an external file that has already been opened, the value of the ACCESS attribute must match the value of the SHARING attribute given to the file when it was first opened (or created). Also, the value of the SHARING attribute must match the value of the ACCESS attribute given to the file when it was first opened (or created). Otherwise, the attempt to open the external file will raise the exception USE_ERROR.
- If you specify any SHARING attribute and do not specify PROHIBIT, then PROHIBIT has no default value (which is equivalent to a default of NO).
- The SHARING attributes are ignored for record-oriented devices and magnetic tape files that are mounted foreign. For ANSI magnetic tape files, a concurrent OPEN operation raises the exception USE_ERROR, even though a SHARING attribute may be specified in the initial OPEN operation. The number of shared files is restricted by the system-wide shared-file database.
- Although write sharing is allowed for all files, you can improve the performance of your program if you avoid write sharing. See the *Guide to VMS File Applications* for more information.

In Example 3–1, read sharing is desired for the relative file REL_FILE.

Example 3–1: Creating and Opening a Relative File for Read Sharing

```
with RELATIVE_IO;
package REL_PKG is new RELATIVE_IO(STRING);

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

with REL_PKG; use REL_PKG;
procedure CREATE_RELATIVE is
   REL_FILE: FILE_TYPE;
    . . .
begin
   CREATE(FILE => REL_FILE,
          MODE => INOUT_FILE,          ❶
          NAME => "REL_FILE.DAT",
          FORM => "RECORD;"      &
                     "SIZE 30;" &
                  "SHARING;"     &     ❷
                     "GET YES;");
 . . .
end CREATE_RELATIVE;
-------------------------------------------------
```

(continued on next page)

Example 3–1 (Cont.): Creating and Opening a Relative File for Read Sharing

```
with REL_PKG; use REL_PKG;
with CREATE_RELATIVE;
procedure SHARE_RELATIVE is
   IO_FILE: FILE_TYPE;
    . . .
begin
   CREATE_RELATIVE;                          ❸
    . . .
   OPEN(FILE => IO_FILE,
        MODE => IN_FILE,
        NAME => "REL_FILE.DAT",              ❹
        FORM => "RECORD;"        &
                   "SIZE 30;"    &
                "SHARING;"       &           ❺
                   "PUT YES;");
   . . .
   CLOSE(IO_FILE);
end SHARE_RELATIVE;
```

Key to Example 3–1:

❶ The CREATE statement creates a relative, in-out file. VAX Ada gives it the following attributes by default (see Table 3–9):

ACCESS; DELETE YES;
ACCESS; GET YES;
ACCESS; PUT YES;
ACCESS; UPDATE YES;
SHARING; GET NO;

Because YES values are in effect for ACCESS DELETE, PUT, and UPDATE, the value of SHARING PROHIBIT is also YES (see Table 3–4).

❷ The CREATE statement specifies a value of YES for SHARING GET; by default, SHARING GET is disallowed and all other sharing is prohibited. SHARING GET indicates that the external file REL_FILE.DAT can be shared with other users who wish to read the file.

❸ The procedure SHARE_RELATIVE calls the procedure CREATE_RELATIVE. Because CREATE_RELATIVE does not close REL_FILE.DAT, the file will still be open and will need to be shared when SHARE_RELATIVE tries to access it.

❹ The OPEN statement opens REL_FILE.DAT as an in file, as only reading is required.

❺ The OPEN statement specifies a value of YES for SHARING PUT, which allows SHARE_RELATIVE to open the external file REL_FILE.DAT. If SHARING PUT is not specified, the file cannot be opened, and the exeception USE_ERROR will be raised.

3.5 Record Locking

The VMS RMS record locking facility allows more than one program to concurrently add, delete, or update a VMS RMS record in a controlled manner. Record locking is available to external files in the same VMS process and across different processes. The *Guide to VMS File Applications* explains VMS RMS record locking in detail.

In VAX Ada, record locking is available for all files. When you open a file for which the attributes SHARING GET, SHARING PUT, or SHARING UPDATE have been specified in the FORM parameter, VMS RMS locks each record as it is accessed. The same external file may then be reopened and associated with another Ada file according to the kind of sharing specified.

When a record of a relative or indexed external file is locked as the result of an operation on a particular Ada file, any other operation on another Ada file that attempts to access the same record will fail, and the exception LOCK_ERROR is raised. When an attempt is made to access a record of any other kind of external file, the exception USE_ERROR is raised. For all files, any subsequent file operation (for example, read, write, modify, delete, end-of-file, and so on) could potentially unlock a previously locked record. See Chapter 14 of the *VAX Ada Language Reference Manual* for descriptions of the effects of the various file operations on locking and unlocking the elements of Ada files.

The following example shows a technique for handling LOCK_ERROR. In this example, attempts to access the record are continued each time a Y (Yes) answer is given to an interactive prompt.

```
-- REL_FILE has been created and opened for read sharing;
-- it is associated with the external file "REL_FILE.DAT".
--
REL_PKG.READ (FILE => REL_FILE,
              ITEM => READ_VALUE,
              FROM => REL_PKG.COUNT(I));
--
-- Additional processing of the record at location COUNT(I)
-- could take place here.
--
 . . .
--
-- IO_FILE has been opened to read the same external file
-- "REL_FILE.DAT". Because both this and the previous READ
-- statement access the same record, potential lock errors
-- could occur.
--
-- Thus, a loop conditionalized on the BOOLEAN variable
-- HAVE_RECORD checks for lock error and issues an interactive
-- prompt if a lock error has occurred. By answering the prompt,
-- the application user can control whether the application
-- waits until the lock is cleared or execution is terminated.
--
while not HAVE_RECORD loop
   begin
      REL_PKG.READ(FILE => IO_FILE,
                   ITEM => READ_VALUE,
                   FROM => REL_PKG.COUNT(I));
      HAVE_RECORD := TRUE;
   exception
      when LOCK_ERROR =>
         TEXT_IO.PUT("Record locked - try again? (Y or N)");
         TEXT_IO.GET(RESPONSE);
         if RESPONSE = "N" then
            raise;                  -- Re-raise LOCK_ERROR.
         end if;
   end;
end loop;
 . . .
```

3.6 Binary Input-Output

VAX Ada provides two kinds of binary input-output packages:

- One kind—SEQUENTIAL_IO, DIRECT_IO, RELATIVE_IO, and INDEXED_IO—allows you to work with binary files containing elements that are all of the same type (a file of elements of an integer type, a file of elements of a record type, a file of elements of an array type, and so on). These packages are all generic; you must instantiate them with the type of the elements in the file before you can use their operations.

- The second kind—SEQUENTIAL_MIXED_IO, DIRECT_MIXED_IO, RELATIVE_MIXED_IO, and INDEXED_MIXED_IO—allows you to work with binary files of mixed types. For example, you can have a mixed-type file that contains elements of three different integer types, or a file that contains elements that are a mixture of integer types, array types, string types, and so on.

The mixed-type packages are nongeneric, but they involve buffer operations that are generic. For example, you must instantiate the generic GET_ITEM and PUT_ITEM operations to move values in and out of a buffer; you then read or write the buffer to transfer a record to or from your file. Example 3–2 and Figure 3–1 show the use of a mixed-type file (using the package DIRECT_MIXED_IO). The circled numbers in Figure 3–1 match statements in the program EXPENSE_ACCOUNT (Example 3–2) to elements in the file EXPENSES. Figure 3–2 shows the use of a file with elements of the same type (using the package DIRECT_IO).

Sections 3.1.1 through 3.1.5 describe the structure of VAX Ada files and give their relationship to VMS RMS files; Chapter 14 of the *VAX Ada Language Reference Manual* describes the packages and their operations in more detail. The following sections give more information (including default file attributes) and present examples that show the features of each kind of package. If you are interested in information about designing files and tuning them for optimum performance, see the *Guide to VMS File Applications*.

Example 3–2: Using a Mixed-Type File

```
with DIRECT_MIXED_IO; use DIRECT_MIXED_IO;
procedure EXPENSE_ACCOUNT is
   type AMOUNT is delta 0.01 range 0.00..5000.00;
   subtype DATE_TYPE is STRING(1..8);
   COUNT: NATURAL := 0;

   procedure PUT_DATE is new PUT_ITEM(DATE_TYPE);
   procedure PUT_COUNT is new PUT_ITEM(NATURAL);
   procedure PUT_COST is new PUT_ITEM(AMOUNT);

   procedure GET_DATE is new GET_ITEM(DATE_TYPE);
   procedure GET_COUNT is new GET_ITEM(NATURAL);
   procedure GET_COST is new GET_ITEM(AMOUNT);

   EXPENSES: FILE_TYPE;
begin
   CREATE(FILE => EXPENSES,
          MODE => INOUT_FILE,
          NAME => "EXPENSES.DAT",
          FORM => "RECORD;"              &
                  "FORMAT FIXED;"        &
                  "SIZE 32;");
   PUT_DATE(EXPENSES, "01-08-84");       (1)
   WRITE(EXPENSES,1);                    (2)
   PUT_COST(EXPENSES, 0.80);             (3)
   COUNT := COUNT + 1;
   PUT_COST(EXPENSES, 27.95);            (4)
   COUNT := COUNT + 1;
   PUT_COST(EXPENSES, 35.00);            (5)
   COUNT := COUNT + 1;
   WRITE(EXPENSES, 3);                   (6)
   PUT_COUNT(EXPENSES, COUNT);           (7)
   WRITE(EXPENSES, 2);                   (8)
   RESET(EXPENSES);
   READ(EXPENSES,2);                     (9)
   GET_COUNT(EXPENSES, COUNT);           (10)
   CLOSE(EXPENSES);
end EXPENSE_ACCOUNT;
```

Figure 3–1: Using a Mixed-Type File

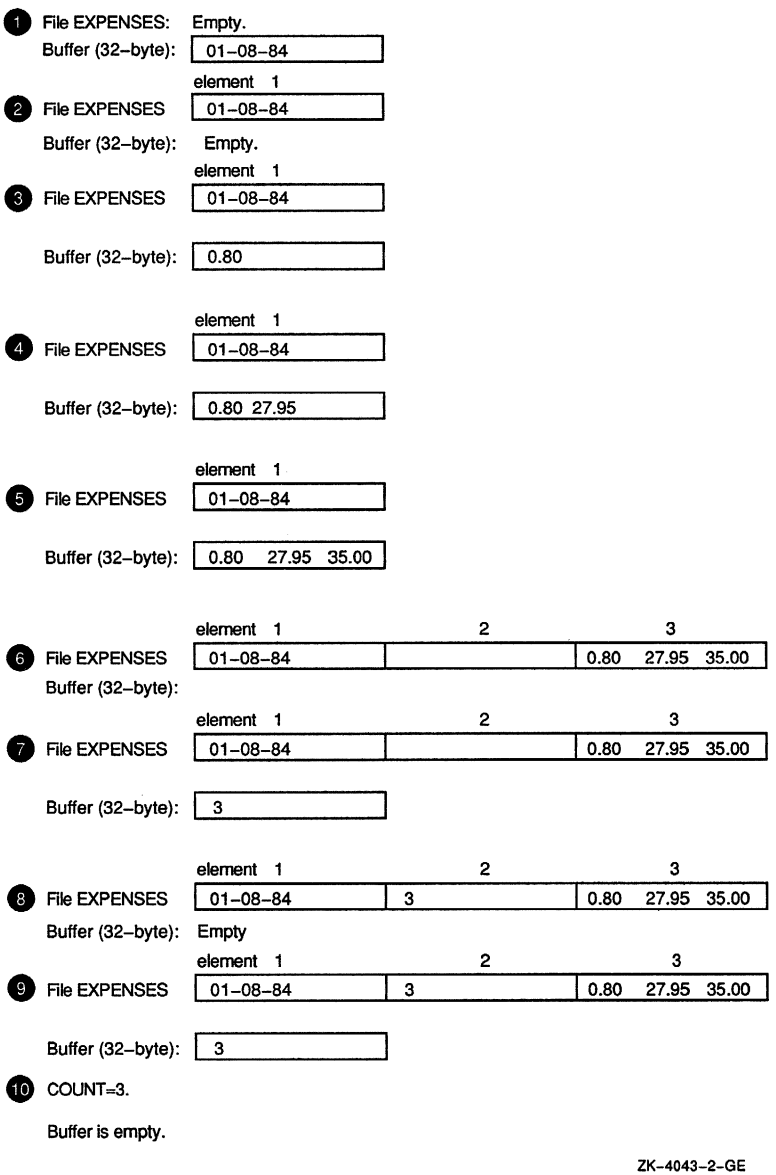
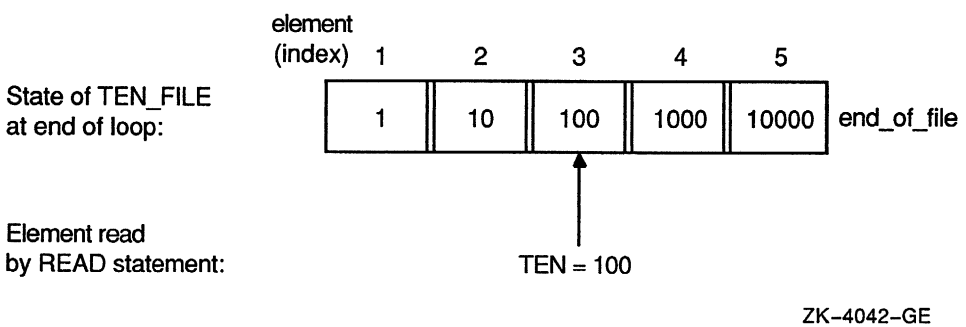


Figure 3–2: Using a Uniform-Type File

```
with DIRECT_IO;
procedure POWERS_OF_TEN is
   package TEN_IO is new DIRECT_IO (NATURAL);
   use TEN_IO;
   TEN: NATURAL := 10;
   POWER: NATURAL;
   TEN_FILE: FILE_TYPE;
begin
   CREATE (TEN_FILE, INOUT_FILE, "TEN_FILE.DAT");
   for POWER in 0..5 loop
       WRITE (TEN_FILE, TEN ** POWER);
   end loop;
   RESET (TEN_FILE);
   READ (TEN_FILE, TEN, 3);
end POWERS_OF_TEN;
```



ZK-4042-GE

3.6.1 Sequential File Input-Output

For creating and working with sequential files of uniform-type elements, VAX Ada provides the generic package SEQUENTIAL_IO; for creating and working with sequential files of mixed-type elements, VAX Ada provides the nongeneric package SEQUENTIAL_MIXED_IO.

When you create a file with the package SEQUENTIAL_IO, VAX Ada gives it the default attributes listed in Table 3–5. When you create a file with the package SEQUENTIAL_MIXED_IO, VAX Ada gives it the default attributes listed in Table 3–6. You can use the operations in the packages

SEQUENTIAL_IO and SEQUENTIAL_MIXED_IO to open and read files of any VMS RMS organization.

Table 3–5: SEQUENTIAL_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	SEQUENTIAL
SEQUENTIAL_ONLY	YES
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	FIXED if ELEMENT_TYPE is constrained; VARIABLE if unconstrained
SIZE	(ELEMENT_TYPE' MACHINE_SIZE + 7)/8 if ELEMENT_TYPE is constrained; 0 (unlimited) if not (note, however, that there are physical limitations to SIZE; see the *VMS Record Management Services Manual*)
ACCESS	
GET	YES
PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
TRUNCATE	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE
CONNECT	
READ_AHEAD	YES
TRUNCATE_ON_PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
WRITE_BEHIND	YES if MODE is OUT_FILE

Table 3–6: SEQUENTIAL_MIXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	SEQUENTIAL
SEQUENTIAL_ONLY	YES
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	VARIABLE
SIZE	0 (record size is unlimited; note, however, that SIZE has physical limitations; see the *VMS Record Management Services Manual*)
ACCESS	
GET	YES
PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
TRUNCATE	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE
CONNECT	
READ_AHEAD	YES
TRUNCATE_ON_PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
WRITE_BEHIND	YES if MODE is OUT_FILE

Example 3–3 shows how to instantiate the package SEQUENTIAL_IO. It also shows how to open, close, read, and write from an Ada sequential file.

The item input-output operations provided by the package SEQUENTIAL_MIXED_IO are basically the same as those provided for the other mixed-type packages. See Figure 3–1, and Examples 3–4 and 3–7 for examples of using the item input-output operations.

Example 3–3: Using the Package SEQUENTIAL_IO

```
with SEQUENTIAL_IO;
procedure SHOW_SEQ is

   type STRING_TYPE is new STRING(1..10);
   package INOUT_STRING is new SEQUENTIAL_IO(STRING_TYPE);
   use INOUT_STRING;

   STRING_FILE : FILE_TYPE;
   STRING_VAR  : STRING_TYPE;

begin

   -- Write a string to the file STRINGDAT.DAT.
   --
   CREATE(FILE => STRING_FILE,
          MODE => OUT_FILE,
          NAME => "STRINGDAT.DAT");
   WRITE (STRING_FILE, "tenletters");
   CLOSE (STRING_FILE);

   -- Read a string from the same file.
   --
   OPEN   (FILE => STRING_FILE,
           MODE => IN_FILE,
           NAME => "STRINGDAT.DAT");
   READ(STRING_FILE,STRING_VAR);
   CLOSE(STRING_FILE);

end SHOW_SEQ;
```

3.6.2 Direct File Input-Output

For creating and working with direct files of uniform-type elements, VAX Ada provides the generic package DIRECT_IO; for creating and working with direct files of mixed-type elements, VAX Ada provides the nongeneric package DIRECT_MIXED_IO.

When you create a file with the package DIRECT_IO, VAX Ada gives it the default file attributes listed in Table 3–7. When you create a file with the package DIRECT_MIXED_IO, VAX Ada gives it the default file attributes listed in Table 3–8. You can use these packages only with files having the FDL attributes ORGANIZATION SEQUENTIAL and RECORD FORMAT FIXED. If you try to use DIRECT_IO or DIRECT_MIXED_IO with a file that has different ORGANIZATION and RECORD FORMAT attributes, the exception USE_ERROR will be raised.

When creating files with the package DIRECT_IO, you must specify a maximum record size with the FORM parameter if you instantiate the package with an unconstrained element type. When creating files with the package DIRECT_MIXED_IO, you must specify a maximum record size with the FORM parameter. The maximum record size determines the maximum size of an element in the file. In the case of DIRECT_MIXED_IO, the maximum record size also determines the size of the file buffer for performing item input-output. If you write a value to a direct file element that is smaller than the size specified, the corresponding external file record is padded with zeros.

Table 3–7: DIRECT_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	SEQUENTIAL
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	FIXED
SIZE	(ELEMENT_TYPE'MACHINE_SIZE + 7)/8 if ELEMENT_TYPE is constrained; otherwise, a value must be specified (no default if ELEMENT_TYPE is unconstrained)
ACCESS	
GET	YES
PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE
CONNECT	
UPDATE_IF	YES

Table 3–8: DIRECT_MIXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	SEQUENTIAL
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	FIXED
SIZE	None; this attribute must be specified in the FORM parameter
ACCESS	
GET	YES
PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE
CONNECT	
UPDATE_IF	YES

Example 3–4 shows the reading and writing of items into a direct file using the package DIRECT_MIXED_IO. For an example of using the package DIRECT_IO, see Figure 3–2.

Note from Example 3–4 that read and write operations to direct files do not have to be to consecutive elements. However, if you read from an empty element, the value returned will be unpredictable.

Example 3–4: Using the Package DIRECT_MIXED_IO

```
with DIRECT_MIXED_IO; use DIRECT_MIXED_IO;
procedure SHOW_DIRECT_MIXED is

   OLD_STRING : STRING(1..5) := "FOUR ";
   NEW_STRING : STRING(1..5) := "FIVE ";
   OLD_INT    : INTEGER := 1;
   NEW_INT    : INTEGER := 4;

   MY_FILE    : FILE_TYPE;

   -- Instantiate the GET and PUT procedures.
   --
   procedure GET_INT is new GET_ITEM(INTEGER);
   procedure GET_STR is new GET_ITEM(STRING);
   procedure PUT_INT is new PUT_ITEM(INTEGER);
   procedure PUT_STR is new PUT_ITEM(STRING);

begin

   -- Create the file; sequential organization is the default,
   -- but is specified for completeness; a record size
   -- must be specified (there is no default).
   --
   CREATE(FILE => MY_FILE,
          MODE => INOUT_FILE,
          NAME => "MY_FILE.DAT",
          FORM => "FILE;"                          &
                     "ORGANIZATION SEQUENTIAL;" &
                  "RECORD;"                        &
                     "SIZE 120;"                    );

   -- Alternately put a string in the buffer and write it
   -- to the file as a single-element record.
   --
   PUT_STR(MY_FILE,OLD_STRING);
   WRITE(FILE => MY_FILE,
         TO   => 1);         -- String will be written to element 1.

   PUT_STR(MY_FILE,OLD_STRING);
   WRITE(FILE => MY_FILE);  -- String will be written to element 2.

   PUT_STR(MY_FILE,OLD_STRING);
   WRITE(FILE => MY_FILE,
         TO   => 5);         -- String will be written to element 5.

   SET_INDEX(MY_FILE, 7);    -- Reposition file pointer to element 7.
   PUT_INT(MY_FILE,OLD_INT);
   WRITE(FILE => MY_FILE);  -- Integer will be written to element 7.
```

(continued on next page)

Example 3–4 (Cont.): Using the Package DIRECT_MIXED_IO

```
   -- Reset for reading.
   --
   RESET(MY_FILE);

   -- Read values from the file.
   --
   READ(MY_FILE);              -- Put the record from element 1
                               -- into the buffer.
   GET_STR(MY_FILE,NEW_STRING);
   READ(FILE => MY_FILE,       -- Put the record from element 7
        FROM => 7);            -- into the buffer.
    . . .
end SHOW_DIRECT_MIXED;
```

3.6.3 Relative File Input-Output

For creating and working with relative files of uniform-type elements, VAX Ada provides the generic package RELATIVE_IO; for creating and working with relative files of mixed-type elements, VAX Ada provides the nongeneric package RELATIVE_MIXED_IO.

When you create a file with the package RELATIVE_IO, VAX Ada gives it the default file attributes listed in Table 3–9. When you create a file with the package RELATIVE_MIXED_IO, VAX Ada gives it the default file attributes listed in Table 3–10. You can use these packages only with files having the attribute ORGANIZATION RELATIVE. If you try to use RELATIVE_IO and RELATIVE_MIXED_IO with a file with any other ORGANIZATION attribute, the exception USE_ERROR will be raised.

When creating files with the package RELATIVE_IO, you must specify a maximum record size with the FORM parameter if you instantiate the package with an unconstrained element type. When creating files with the package RELATIVE_MIXED_IO, you must specify a maximum record size with the FORM parameter. The maximum record size determines the maximum size of an element in the file. In the case of RELATIVE_MIXED_IO, the maximum record size also determines the size of the file buffer for performing item input-output.

Table 3–9: RELATIVE_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	RELATIVE
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	FIXED if ELEMENT_TYPE is constrained; VARIABLE if not
SIZE	(ELEMENT_TYPE' MACHINE_SIZE + 7)/8 if ELEMENT_TYPE is constrained; if not, a value must be specified (there is no default if ELEMENT_TYPE is unconstrained)
ACCESS	
DELETE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
GET	YES
PUT	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
UPDATE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE or INOUT_FILE

Table 3–10: RELATIVE_MIXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	RELATIVE
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	VARIABLE

(continued on next page)

Table 3–10 (Cont.): RELATIVE_MIXED_IO: Default File Attributes

File Attribute	Default Value
SIZE	None; a value must be specified in the FORM parameter
ACCESS	
DELETE	YES if MODE is OUT_FILE or INOUT_FILE; NO if mode is IN_FILE
GET	YES
PUT	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
UPDATE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE or INOUT_FILE

Example 3–5 shows the reading and writing of records to cells in a relative file using the package RELATIVE_IO. Read and write operations to relative files do not have to be to consecutively numbered; however, if you try to read at a position for which there is no element, the exception EXISTENCE_ERROR will be raised.

The item input-output operations provided by the package RELATIVE_MIXED_IO are basically the same as those provided for the other mixed-type packages. See Figure 3–1 and Examples 3–4 and 3–7 for examples of using the item input-output operations.

Example 3–5: Using the Package RELATIVE_IO

```
with RELATIVE_IO;
procedure SHOW_RELATIVE_IO is

   type SMALL_RECORD is
      record
         NUM: INTEGER := 0;
         LET: CHARACTER := 'A';
      end record;

   -- Instantiate and make visible a RELATIVE_IO package
   -- that operates on elements of type SMALL_RECORD.
   --
   package REC_IO is new RELATIVE_IO(SMALL_RECORD);
   use REC_IO;

   -- Declare the objects to be used.
   --
   RELATIVE_FILE : FILE_TYPE;
   POS           : POSITIVE_COUNT;
   REC           : SMALL_RECORD;
   RECX          : SMALL_RECORD := (NUM => 1, LET => 'X');
   RECY          : SMALL_RECORD := (NUM => 2, LET => 'Y');
   I             : INTEGER;

begin

   -- Create the file.
   --
   CREATE(RELATIVE_FILE,OUT_FILE,"RELATIVE_FILE.DAT");

   -- Write records, incrementing the NUM value, to file
   -- cells in positions 1 through 10.
   --
   for I in 1..10 loop
      WRITE(RELATIVE_FILE,REC);
      REC.NUM := REC.NUM + 1;
   end loop;

   -- Prepare the file for reading.
   --
   RESET(RELATIVE_FILE,IN_FILE);

   -- Read contents of records in cells at positions 2 and 3.
   --
   POS := INDEX(RELATIVE_FILE);
   READ(RELATIVE_FILE,RECX,2);
   POS := INDEX(RELATIVE_FILE);
   READ(RELATIVE_FILE,RECY);
```

(continued on next page)

Example 3–5 (Cont.): Using the Package RELATIVE_IO

```
   -- Prepare the file for writing.
   --
   RESET(RELATIVE_FILE,OUT_FILE);

   -- Write to records in cells at positions 12 and 16.
   --
   WRITE(RELATIVE_FILE,REC,12);
   REC.NUM := REC.NUM + 1;
   WRITE(RELATIVE_FILE,REC,16);
    . . .
end SHOW_RELATIVE_IO;
```

3.6.4 Indexed File Input-Output

For creating and working with indexed files of uniform-type elements, VAX Ada provides the generic package INDEXED_IO; for creating and working with indexed files of mixed-type elements, VAX Ada provides the nongeneric package INDEXED_MIXED_IO.

When you create a file with the package INDEXED_IO, VAX Ada gives it the default file attributes listed in Table 3–11. When you create a file with the package INDEXED_MIXED_IO, VAX Ada gives it the default file attributes listed in Table 3–12. You can use these packages only with files having the attribute ORGANIZATION INDEXED. If you try to use INDEXED_IO or INDEXED_MIXED_IO with a file that has a different ORGANIZATION attribute, the exception USE_ERROR will be raised.

When creating indexed files, you must use the FORM parameter to specify any information about the keys (no default key values are provided by the CREATE procedures). Note that there is no default bucket size; if you do not specify a bucket size with the FORM parameter, VMS RMS calculates the bucket size based on the maximum record size (the default is 0).

Table 3–11: INDEXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	INDEXED
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	FIXED if ELEMENT_TYPE is constrained; VARIABLE if not
SIZE	(ELEMENT_TYPE' MACHINE_SIZE + 7)/8 if ELEMENT_TYPE is constrained; 0 if not (there is no maximum record size; note, however, that SIZE is also limited by the bucket size; see the *VMS Record Management Services Manual*)
ACCESS	
DELETE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
GET	YES
PUT	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
UPDATE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE or INOUT_FILE

Table 3–12: INDEXED_MIXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	INDEXED
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	VARIABLE

(continued on next page)

Table 3–12 (Cont.): INDEXED_MIXED_IO: Default File Attributes

File Attribute	Default Value
SIZE	0 (the record size is unlimited; note, however, that the record size is limited by the bucket size; see the *VMS Record Management Services Manual*)
ACCESS	
DELETE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
GET	YES
PUT	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
UPDATE	YES if MODE is OUT_FILE or INOUT_FILE; NO if MODE is IN_FILE
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE or INOUT_FILE

You can access indexed files with both sequential and keyed access methods. Sequential access retrieves consecutive components, which are sorted according to the specified key field. Keyed access retrieves components randomly, according to the value of a particular key field. Once you select a key (using the RESET or READ_BY_KEY procedures), a sequential read (using the READ procedure) retrieves components with ascending or descending key field values.

Example 3–6 shows the use of the package INDEXED_IO to create an indexed file that has a string-type primary key that sorts the file in ascending order, and a string-type alternate key that sorts the file in descending order. In particular, the example shows how to do comparative key searching in an indexed file.

In VAX Ada, the way to do comparative key searching is to use the indexed input-output package READ_BY_KEY procedures (see Chapter 14 of the *VAX Ada Language Reference Manual* for their specifications). The kind of comparison (equal or next, equal, or next) is determined by the value of the READ_BY_KEY RELATION parameter. The parameter is of the type RELATION_TYPE, and its default value for both packages INDEXED_IO and INDEXED_MIXED_IO is EQUAL. The value of a READ_BY_KEY RELATION parameter overrides any search option setting you may have made in a CREATE or OPEN FORM parameter. In other words, the FDL

CONNECT EQUAL_NEXT and CONNECT_NEXT attributes never have an effect when you are using a READ_BY_KEY procedure.

Example 3–6: Using the Package INDEXED_IO

```
-- Create an INDEXED_IO package for indexed files containing
-- string data.
--
with INDEXED_IO;
package STRING_INDEXED_IO is new INDEXED_IO (STRING);

with TEXT_IO; use TEXT_IO;
with STRING_INDEXED_IO; use STRING_INDEXED_IO;
procedure SHOW_INDEX is

    IFILE   : STRING_INDEXED_IO.FILE_TYPE;
    STR     : STRING (1..10) := "          ";
    KEY_STR : STRING (1..1);

    -- Instantiate generic READ_BY_KEY procedure for ascending
    -- string matching (as opposed to numeric key matching).
    --
    procedure READ_BY_STRING_KEY is new READ_BY_KEY (STRING, 0);

begin

    PUT_LINE ("-- Test of INDEXED_IO.");
    PUT_LINE ("-- Creating file");

    -- The CREATE procedure must give key information. KEY 0 has
    -- ascending sort order; KEY 1 has descending -- the sort
    -- order is determined by the value of the KEY TYPE
    -- attributes in the form string: STRING or DSTRING. (Do
    -- not confuse this STRING with the Ada type STRING.)
    --
```

(continued on next page)

Example 3–6 (Cont.): Using the Package INDEXED_IO

```
-- Because this is an indexed file of the Ada type STRING, and
-- the Ada type STRING is an unconstrained type, you must
-- also specify the maximum record size. A size of 0 bytes
-- is used so that the system will not impose a maximum
-- record length.
--
CREATE (FILE => IFILE,
        MODE => INOUT_FILE,
        NAME => "INDEXED_STRING.TXT",
        FORM => "FILE;"                          &
                   "ORGANIZATION INDEXED;"       &
                "RECORD;"                        &
                    "SIZE 0;"                    &
                "KEY 0;"                         &
                    -- Key value STRING causes
                    -- ascending sort.
                    "TYPE STRING;"               &
                    "POSITION 0;"                &
                    "LENGTH 1;"                  &
                    "DUPLICATES YES;"            &
                "KEY 1;"                         &
                    -- Key value DSTRING causes
                    -- descending sort.
                    "TYPE DSTRING;"              &
                    "POSITION 0;"                &
                    "LENGTH 1;"                  &
                    "DUPLICATES YES;"            );

-- Populate file.
--
PUT_LINE ("-- Populating file");
WRITE (IFILE, "Mary      ");
WRITE (IFILE, "Larry     ");
WRITE (IFILE, "Charlie   ");
WRITE (IFILE, "Kirk      ");
WRITE (IFILE, "Spencer   ");
WRITE (IFILE, "Susan     ");
```

(continued on next page)

Example 3–6 (Cont.): Using the Package INDEXED_IO

```
-- Read file sequentially using ascending index.
--
PUT_LINE ("-- Read file sequentially: ascending sort");
RESET (FILE => IFILE ,
       MODE => INOUT_FILE,
       KEY_NUMBER => 0);
while not END_OF_FILE(IFILE)
   loop
      READ (IFILE, STR);
      PUT_LINE (STR);
   end loop;

-- Read file sequentially using descending index.
--
PUT_LINE ("-- Read file sequentially: descending sort");
RESET (FILE       => IFILE,
       MODE       => INOUT_FILE,
       KEY_NUMBER => 1);
while not END_OF_FILE(IFILE)
    loop
        READ (IFILE, STR);
        PUT_LINE (STR);
    end loop;

--
-- Change the search to EQUAL_NEXT using the instantiation
-- of READ_BY_KEY (READ_BY_STRING_KEY), and read the whole
-- file by ascending key.
--
PUT_LINE ("-- READ_BY_KEY: ascending index");
RESET (FILE => IFILE);
KEY_STR := "M";

-- Read the first item that is equal to or that follows a string
-- whose first character is "M". Use READ_BY_STRING_KEY to
-- set the character match, key number (0 in this example
-- translates to an ascending key), and relation.
--
READ_BY_STRING_KEY (FILE => IFILE,
                    ITEM => STR,
                    KEY  => KEY_STR,
                    KEY_NUMBER => 0,
                    RELATION => EQUAL_NEXT);
PUT_LINE (STR);
```

(continued on next page)

Example 3–6 (Cont.): Using the Package INDEXED_IO

```
    -- Read the rest of the strings that meet the
    -- requirements specified in the READ_BY_STRING_KEY statement
    -- using READ (a loop of READ_BY_KEY will endlessly
    -- return the first match).
    --
    while not END_OF_FILE(IFILE)
        loop
            READ (IFILE, STR);
            PUT_LINE (STR);
        end loop;

    -- Read by descending key only those records that begin
    -- with "S". Use READ_BY_STRING_KEY to set the character
    -- match, key number (1 in this example translates to a
    -- descending key), and relation.
    --
    PUT_LINE ("-- READ_BY_KEY: descending index");
    RESET (FILE => IFILE);
    KEY_STR := "S";
    READ_BY_STRING_KEY (FILE => IFILE,
                        ITEM => STR,
                        KEY  => KEY_STR,
                        KEY_NUMBER => 1,
                        RELATION => EQUAL);
    PUT_LINE (STR);
    while not END_OF_FILE(IFILE)
        loop
            READ (IFILE, STR);
            PUT_LINE (STR);
        end loop;

    -- Finish.
    --
    PUT_LINE ("-- Closing file");
    CLOSE (FILE => IFILE );

end SHOW_INDEX;
```

Example 3–7 shows the use of the package INDEXED_MIXED_IO, shows how to create a mixed-type indexed file, and then shows how to read and write from the file using the primary key.

Example 3–7: Using the Package INDEXED_MIXED_IO

```
with INDEXED_MIXED_IO; use INDEXED_MIXED_IO;
procedure SHOW_INDEXED_MIXED is

   type INTEGER_ARRAY_TYPE is array(INTEGER range <>) of INTEGER;
   type COLORS is (RED,BLUE,YELLOW);

   -- Declare objects to be used to fill the file with values.
   --
   INDEXED_FILE           : FILE_TYPE;
   INTEGER_ARRAY          : INTEGER_ARRAY_TYPE(1..3);
   INT1, INT2, INT3,
   INT4, INT5, INT6, INT7 : INTEGER;
   CHAR1, CHAR2,
   CHAR3, CHAR4           : CHARACTER;
   COL1, COL2             : COLORS;
   ARRAY_INDEX            : INTEGER;

   -- Instantiate the generic READ_BY_KEY procedures.
   --
   procedure READ_0 is new READ_BY_KEY(INTEGER,0);
   procedure READ_1 is new READ_BY_KEY(CHARACTER,1);

   -- Instantiate the generic GET_ITEM and PUT_ITEM procedures.
   --
   procedure GET_INT   is new GET_ITEM(INTEGER);
   procedure GET_FLOAT is new GET_ITEM(FLOAT);
   procedure GET_CHAR  is new GET_ITEM(CHARACTER);
   procedure GET_ENUM  is new GET_ITEM(COLORS);

   procedure PUT_INT   is new PUT_ITEM(INTEGER);
   procedure PUT_FLOAT is new PUT_ITEM(FLOAT);
   procedure PUT_CHAR  is new PUT_ITEM(CHARACTER);
   procedure PUT_ENUM  is new PUT_ITEM(COLORS);

   procedure GET_ARRAY_INT is new
      GET_ARRAY(INTEGER,INTEGER,INTEGER_ARRAY_TYPE);
```

(continued on next page)

Example 3–7 (Cont.): Using the Package INDEXED_MIXED_IO

```
begin

-- Create the file.
--
   CREATE(FILE => INDEXED_FILE,
          MODE => OUT_FILE,
          NAME => "F.DAT",
          FORM => "FILE;"                        &
                     "ORGANIZATION INDEXED;"     &
                  "KEY 0;"                       &
                     "INDEX_FILL 4;"             &
                     "TYPE INT4;"                &
                     "DUPLICATES YES;"           &
                     "POSITION 0;"               &
                     "LENGTH 4;"                 &
                  "KEY 1;"                       &
                     "INDEX_FILL 1;"             &
                     "TYPE STRING;"              &
                     "DUPLICATES YES;"           &
                     "POSITION 4;"               &
                     "LENGTH 1;"                  );

-- Fill the element buffer with a character, an integer,
-- and an enumeration value.
--
   INT1  := 1;
   CHAR1 := 'A';
   COL1  := YELLOW;
   PUT_INT(INDEXED_FILE,INT1);
   PUT_CHAR(INDEXED_FILE,CHAR1);
   PUT_ENUM(INDEXED_FILE,COL1);

-- Write the element to the file.
--
   WRITE(INDEXED_FILE);

-- Prepare to read the record from the file.
--
   RESET(INDEXED_FILE,INOUT_FILE);

-- Read the record from the file sorting on
-- the primary key (integer).
--
   READ_0(INDEXED_FILE,INT1,0);
   GET_INT(INDEXED_FILE,INT2);
   GET_CHAR(INDEXED_FILE,CHAR2);
   GET_ENUM(INDEXED_FILE,COL2);
```

(continued on next page)

Example 3–7 (Cont.): Using the Package INDEXED_MIXED_IO

```
-- Prepare to add more elements to the file.
--
   RESET(INDEXED_FILE);
   SET_POSITION(INDEXED_FILE,1);

-- Fill the buffer with an integer, a character,
-- and three more integers, and write the buffer to
-- the file.
--
   INT3  := 3;
   CHAR3 := 'B';
   INT4  := 4;
   INT5  := 5;
   INT6  := 6;

   PUT_INT(INDEXED_FILE,INT3);
   PUT_CHAR(INDEXED_FILE,CHAR3);
   PUT_INT(INDEXED_FILE,INT4);
   PUT_INT(INDEXED_FILE,INT5);
   PUT_INT(INDEXED_FILE,INT6);

   WRITE(INDEXED_FILE);

-- Read the record from the file sorting on
-- key 1 (string).
--
   READ_1(INDEXED_FILE,CHAR3,1);

-- Get the items from the buffer; in particular, read
-- three integers directly into the integer array.
--
   GET_INT(INDEXED_FILE,INT7);
   GET_CHAR(INDEXED_FILE,CHAR4);
   GET_ARRAY_INT(INDEXED_FILE,INTEGER_ARRAY,ARRAY_INDEX);

-- Do some more work and then close the file.
--
    . . .
   CLOSE(INDEXED_FILE);

end SHOW_INDEXED_MIXED;
```

3.7 Text Input-Output

VAX Ada provides the package TEXT_IO for creating and working with text files. TEXT_IO is not generic, but it does include generic packages for the input and output of integers, floating-point numbers, fixed-point numbers,

and enumeration values. When you create a file with this package, VAX Ada gives it the defaults listed in Table 3–13.

You can use this package only with files that have the attribute ORGANIZATION SEQUENTIAL. For example, you can use TEXT_IO operations to open and read files created with the packages SEQUENTIAL_IO, SEQUENTIAL_MIXED_IO, DIRECT_IO, or DIRECT_MIXED_IO, as well as TEXT_IO. If you try to use this package with files that have a different ORGANIZATION attribute, the exception USE_ERROR will be raised.

Table 3–13: TEXT_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	SEQUENTIAL
SEQUENTIAL_ONLY	YES
RECORD	
CARRIAGE_CONTROL	PRINT if device is a terminal; CARRIAGE_RETURN otherwise
FORMAT	VFC if device is a terminal; VARIABLE otherwise
SIZE	0 (record size is unlimited; note, however, that the record size has physical limitations; see the *VMS Record Management Services Manual*)
ACCESS	
GET	YES
PUT	YES if MODE is OUT_FILE; NO if MODE is IN_FILE
TRUNCATE	YES if MODE is OUT_FILE; NO if MODE is IN_FILE

(continued on next page)

Table 3–13 (Cont.): TEXT_IO: Default File Attributes

File Attribute	Default Value
SHARING	
GET	YES if MODE is IN_FILE; NO if MODE is OUT_FILE
CONNECT	
READ_AHEAD	YES
WRITE_BEHIND	YES if MODE is OUT_FILE

As shown in Table 3–13, VAX Ada text files are implemented as VMS RMS sequential files. Each line in a text file corresponds to a single VMS RMS record; VAX Ada text files are not stream files.

Although VAX Ada creates text files with variable-length records by default, you can use the FORM parameter (see Section 3.3) to create text files with fixed-length records. When a text file with fixed-length records is being written, the line length (if nonzero) must be less than or equal to the record size. The exception USE_ERROR is raised if you try to change the line length to a value greater than the record size. This exception is also raised when a line being written is longer than the record size. When you write a program that creates text files with fixed-length records, set the line length to the record size. If the line being written does not fill the entire (fixed-length) record, spaces are used to pad the rest of the record (and the spaces are then regarded as characters in the file).

3.7.1 Using the Package TEXT_IO for Terminal Input-Output

When using the package TEXT_IO to read from or write to a terminal, keep the following points in mind:

- VAX Ada TEXT_IO operations are implemented with VMS RMS input-output operations, and VMS RMS operations always involve complete records.
- Buffering is used in both terminal input and output (see Section 3.7.3).
- Terminal input is not processed until a line (a VMS RMS record) is terminated by a carriage return (or other line terminator).

- CTRL/Z is interpreted sometimes as a file terminator, and sometimes as a line terminator followed by a page terminator followed by a file terminator (the importance and interpretation of the various terminators is discussed in Section 3.7.2). The difference in interpretation can cause a difference in effect.
- You can achieve asynchronous input-output in tasking programs by defining the logical names ADA$INPUT and ADA$OUTPUT so that they refer to nonprocess-permanent files; for example, by defining ADA$INPUT and ADA$OUTPUT so that they refer to TT, you can achieve asynchronous terminal input-output. See Section 3.9.2 for more information.

Example 3–8 shows the use of TEXT_IO operations to write text from a terminal to a file. Sections 3.7.1.1 to 3.7.1.4 discuss a number of coding methods for accomplishing interactive terminal input-output.

Example 3–8: Using the Package TEXT_IO

```
with TEXT_IO; use TEXT_IO;
procedure COPY is

   MY_COPY      : FILE_TYPE;
   INPUT_80     : STRING (1..80);
   CURRENT_PAGE : POSITIVE_COUNT;
   LAST         : NATURAL;

begin

   CREATE(MY_COPY, OUT_FILE, "MY_COPY.TXT");
   PUT_LINE("Start typing your book.");
   PUT_LINE("Type CTRL/Z to finish.");

   loop
      -- Remember current page, then get at most
      -- 80 characters, then write out the line
      -- to the text file.
      --
      CURRENT_PAGE := PAGE (CURRENT_INPUT);
      GET_LINE (INPUT_80, LAST);
      PUT (MY_COPY, INPUT_80(1..LAST));
```

(continued on next page)

Example 3–8 (Cont.): Using the Package TEXT_IO

```
      -- If a new page is started, then terminate
      -- the page in the file. Do not write an explicit
      -- end-of-page if the page change is a result of
      -- an end-of-file (CTRL/Z).  Otherwise, start
      -- a new line.

      if CURRENT_PAGE < PAGE (CURRENT_INPUT) then
         if not END_OF_FILE then
            NEW_PAGE (MY_COPY);
         end if;
      else
         NEW_LINE (MY_COPY);
      end if;
   end loop;

exception
   when END_ERROR =>
      NEW_LINE (3);
      PUT ("Your text is in file MY_COPY.TXT");
      CLOSE (MY_COPY);

end COPY;
```

When working with text input-output in general, and with terminal input-output in particular, keep in mind that each VAX Ada TEXT_IO operation behaves exactly as it is described in the *VAX Ada Language Reference Manual*. For example:

```
with TEXT_IO; use TEXT_IO;
procedure SHOW_GETS is
   INOUT_LINE: STRING(1..10) := "tenletters";
   LAST_CHAR: NATURAL;
begin
   PUT_LINE("Do a GET_LINE");
   GET_LINE(INOUT_LINE,LAST_CHAR);
   PUT_LINE(INOUT_LINE);
   PUT_LINE("Do another GET_LINE");
   GET_LINE(INOUT_LINE,LAST_CHAR);
   PUT(INOUT_LINE);
end SHOW_GETS;
```

If you run this program and press CTRL/Z as the only input to the GET_LINE operation, the immediate result is that the VMS exit prompt appears on your screen, and then the string "tenletters" is printed. This result occurs because GET_LINE is defined as a procedure that replaces the characters of its string argument with input characters until it encounters a line terminator.

Because CTRL/Z in this case represents a line terminator followed by a page terminator followed by a file terminator (see Section 3.7.2), GET_LINE immediately encounters a line terminator. Then, according to the language definition of GET_LINE, SKIP_LINE is called, and the subsequent page terminator is skipped. The initial string is output because it was not changed by GET_LINE. Because the file terminator remains as input for the next GET_LINE operation, the exception END_ERROR is raised when the next GET_LINE operation is executed. If the first GET_LINE had been a GET, the exception END_ERROR would have been raised immediately.

Similarly, if you use the GET_LINE procedure to read a value into a string variable of N characters, and you enter exactly N characters followed by a carriage return, the END_OF_LINE function will return the value FALSE. However, another call to GET_LINE will read in a null string, indicating that there was a line terminator in the input buffer (the carriage return), which was entered after the N characters were entered. This effect occurs because when you read in exactly as many characters as are on the line, the SKIP_LINE procedure is not called after the characters are transferred. The effect is in accordance with the description of the GET_LINE procedure in the *VAX Ada Language Reference Manual.*

You should also be aware that when you do a SKIP_LINE operation in VAX Ada (or any operation that, in effect, does a SKIP_LINE, such as a GET_LINE; see Chapter 14 of the *VAX Ada Language Reference Manual*), the skipping of the page terminator (if any) is delayed. A subsequent operation may require that the skipped page terminator be retrieved, and the result is a request for more input from the file. This delaying process enables a GET_LINE operation from a terminal device to be (partially) satisfied immediately after a carriage return and then for execution of the program to continue.

3.7.1.1 Line-Oriented Method

Example 3–9 shows a line-oriented method of using TEXT_IO operations for interactive terminal input-output. Arbitrary lines are obtained using the procedure GET_LINE within a loop. The actual interpretation of data on each line is deferred to other code, so this method is very flexible and adaptable. The method expects the user to enter one of the following:

- A line of data
- A null line (carriage return)
- An end-of-file indicator (CTRL/Z)

If you want to allow the user to respond with multiple CTRL/Zs, you need to declare a file variable to serve as the input file, rather than using the default standard input file. You need to use a file variable because the only way to get past the first CTRL/Z is to reset the file, and you cannot pass the standard input file as a parameter to the procedure RESET (RESET's file parameter has a mode of **in out**; the standard input file can be used only with a mode of **in**). Example 3–9 declares the variable TERMINAL for this purpose.

Example 3–9 can be extended to obtain whatever data is on each line by using those TEXT_IO operations that read data from a string (in this case, the string variable LINE).

After trying Example 3–9, note that a CTRL/Z is interpreted sometimes as a file terminator and sometimes as a line terminator followed by a page terminator followed by a file terminator. The simplest explanation for this follows:

- CTRL/Z requires a prior line.
- If there is a prior line, the CTRL/Z is interpreted as a file terminator.
- If there is no prior line, the CTRL/Z inserts a null line, and is interpreted as a line terminator followed by a page terminator followed by a file terminator.

In other words, a call to GET_LINE that encounters a CTRL/Z may or may not return a null line before resulting in an END_ERROR.

Example 3–9: Example of Line-Oriented TEXT_IO

```
with TEXT_IO; use TEXT_IO;
procedure IO_EXAMPLE is

   -- This example shows how to input a command line from a
   -- terminal. It shows how to prompt using PUT followed by GET,
   -- and shows how to recover from END_ERROR (CTRL/Z).
   --
   -- To run this program, you must define the logical name
   -- USER_INPUT to point to your terminal. For example:
   --
   --     $ DEFINE USER_INPUT TT
   --
   TERMINAL : FILE_TYPE;
   subtype LINE_TYPE is STRING(1..132);
   LEN      : NATURAL;
   LINE     : LINE_TYPE;

begin

   PUT_LINE("This example is programmed so that entering");
   PUT_LINE("a RETURN or CTRL/Z is ignored.");
   PUT_LINE("All other entries are echoed.");
   PUT_LINE("To quit, type Q or q.");

   -- NOTE: To recover from CTRL/Z (end-of-file) on a terminal, you
   -- must do a RESET. To do a RESET, you must have a file variable.
   -- Thus, you must open the file so that it "speaks" to the
   -- terminal. You cannot use the standard input file (ADA$INPUT)
   -- as the file because RESET takes an 'in out' file as a
   -- parameter, and the standard input file can be used only as an
   -- 'in' parameter.
   --
   -- This example uses the file variable TERMINAL. When TERMINAL is
   -- opened, it is associated with the external file "USER_INPUT:",
   -- which you have defined as a logical name that points to the
   -- terminal. The file variable TERMINAL can be used as an actual
   -- parameter to the RESET procedure.
   --
   OPEN(TERMINAL, IN_FILE, "USER_INPUT:");
   loop
      begin
         -- Note that calls to PUT are buffered until a NEW_LINE or
         -- a GET is entered from the same device. Thus, the
         -- sequence 'PUT GET' results in prompting.
         --
         PUT("Command> ");
         GET_LINE(TERMINAL, LINE, LEN);
```

(continued on next page)

Example 3–9 (Cont.): Example of Line-Oriented TEXT_IO

```
         if LEN = 0 then
            PUT_LINE("Thank you for entering a null line.");
         else
            PUT_LINE("Thank you for entering the command " &
                     LINE(1..LEN));
            if LINE(1..LEN) = "q" or LINE(1..LEN) = "Q" then
               PUT_LINE("Exiting now...");
               exit;
            end if;
         end if;
      exception
         when END_ERROR =>
            RESET(TERMINAL);
            PUT_LINE("Thank you for entering a CTRL/Z.");
      end;
   end loop;

end IO_EXAMPLE;
```

3.7.1.2 Data-Oriented Method

Example 3–10 shows a data-oriented method of using TEXT_IO operations. A sequence of data values is obtained using a series of calls to the GET procedure within a loop. The interpretation of the data is important and embedded in the code that does the input-output, but how the data is laid out across lines is not important. The user is expected to enter one data value (not necessarily a line) at a time. If the wrong kind of data is entered, the exception DATA_ERROR is raised.

Example 3–10: Example of Data-Oriented TEXT_IO

```
with TEXT_IO; use TEXT_IO;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure ANOTHER_IO_EXAMPLE is

   TERMINAL    : FILE_TYPE;

   FLT1_VALUE,
   FLT2_VALUE,
   FLT3_VALUE  : FLOAT;

   INT1_VALUE,
   INT2_VALUE,
   INT3_VALUE  : INTEGER;

begin

   PUT_LINE("This example is programmed so that entering");
   PUT_LINE("a RETURN or CTRL/Z is ignored.");
   PUT_LINE("All other entries are echoed.");
   OPEN(TERMINAL, IN_FILE, "USER_INPUT:");

   loop
      begin
         PUT("Enter 3 integers on arbitrary lines");
         PUT("(to quit enter 0)");

         GET(TERMINAL, INT1_VALUE);
         exit when INT1_VALUE = 0;
         GET(TERMINAL, INT2_VALUE);
         GET(TERMINAL, INT3_VALUE);

         PUT("Ok, we got: ");
         PUT(INT1_VALUE);
         PUT(INT2_VALUE);
         PUT(INT3_VALUE);
         NEW_LINE;
```

(continued on next page)

Example 3–10 (Cont.): Example of Data-Oriented TEXT_IO

```
        PUT("Enter 3 floats on arbitrary lines");
        PUT("(to quit enter 0.0)");

        GET(TERMINAL, FLT1_VALUE);
        exit when FLT1_VALUE = 0.0;
        GET(TERMINAL, FLT2_VALUE);
        GET(TERMINAL, FLT3_VALUE);

        PUT("Ok, we got: ");
        PUT(FLT1_VALUE);
        PUT(FLT2_VALUE);
        PUT(FLT3_VALUE);
        NEW_LINE;

      exception
        when END_ERROR =>
          RESET(TERMINAL);
          PUT_LINE("Ok, let's try again");
      end;
   end loop;

end ANOTHER_IO_EXAMPLE;
```

3.7.1.3 Mixed Method

The mixed method of using TEXT_IO operations sometimes obtains whole lines using the GET_LINE procedure, and sometimes obtains individual data values using the GET procedure. This method is much trickier than the line-oriented or data-oriented method because GET and GET_LINE treat line terminators differently:

- GET skips leading line terminators before reading data.
- GET_LINE (usually) skips line terminators after reading data.

Thus, if you follow a GET with a GET_LINE, the GET_LINE is likely to return a null string found at the end of the current line.

To make GET and GET_LINE compatible, you need to follow the last GET on every line with a SKIP_LINE. However, the SKIP_LINE will ignore any data that the user may have typed after the GET.

The incompatible nature of GET and GET_LINE makes this style complicated and error-prone.

3.7.1.4 Flexible Method

In some cases, you may want to mix the kinds of data the user can enter. For example, you may want to allow users to enter integers where real numbers are normally expected; that is, to enter 3 when 3.0 is expected. You can accomplish this by handling the exception DATA_ERROR as follows:

- Try to read a real number.
- If DATA_ERROR is raised, handle it by trying to read an integer.

Example 3–11 shows the use of this method. The example also shows how you can display a default value that will be used if the user enters no data (a carriage return or CTRL/Z).

NOTE

When you enter a CTRL/Z after entering a line that ends with a carriage return, the CTRL/Z is considered to be the end-of-file. A sequence of two CTRL/Zs is equivalent to the sequence RETURN CTRL/Z.

Example 3–11: Example of Flexible TEXT_IO

```
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with LONG_FLOAT_TEXT_IO; use LONG_FLOAT_TEXT_IO;
procedure GET_NUM (INPUT: in out FILE_TYPE; X: in out LONG_FLOAT) is

   NUM    : INTEGER;
   subtype LINE_TYPE is STRING(1..132);
   LINE   : LINE_TYPE;
   L, LAST: INTEGER;
```

(continued on next page)

Example 3–11 (Cont.): Example of Flexible TEXT_IO

```
begin
   PUT(" [");
   PUT(X,3,2,0);
   PUT("]: ");
   loop
      begin
         GET_LINE(INPUT, LINE, L);
         exit when L = 0;
         GET(LINE(1..L),X,LAST);
         exit;
      exception
         when END_ERROR =>
            RESET(INPUT);
            exit;
         when DATA_ERROR =>
            begin
               GET(LINE(1..L),NUM,LAST);
               X := LONG_FLOAT(NUM);
               exit;
            exception
               when DATA_ERROR =>
                  PUT("  Invalid data, try again: ");
            end;
      end;
   end loop;
end GET_NUM;

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

with GET_NUM;
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with LONG_FLOAT_TEXT_IO; use LONG_FLOAT_TEXT_IO;
procedure THIRD_IO_EXAMPLE is
   NUM   : LONG_FLOAT := 1.0;
   INPUT: TEXT_IO.FILE_TYPE;
```

(continued on next page)

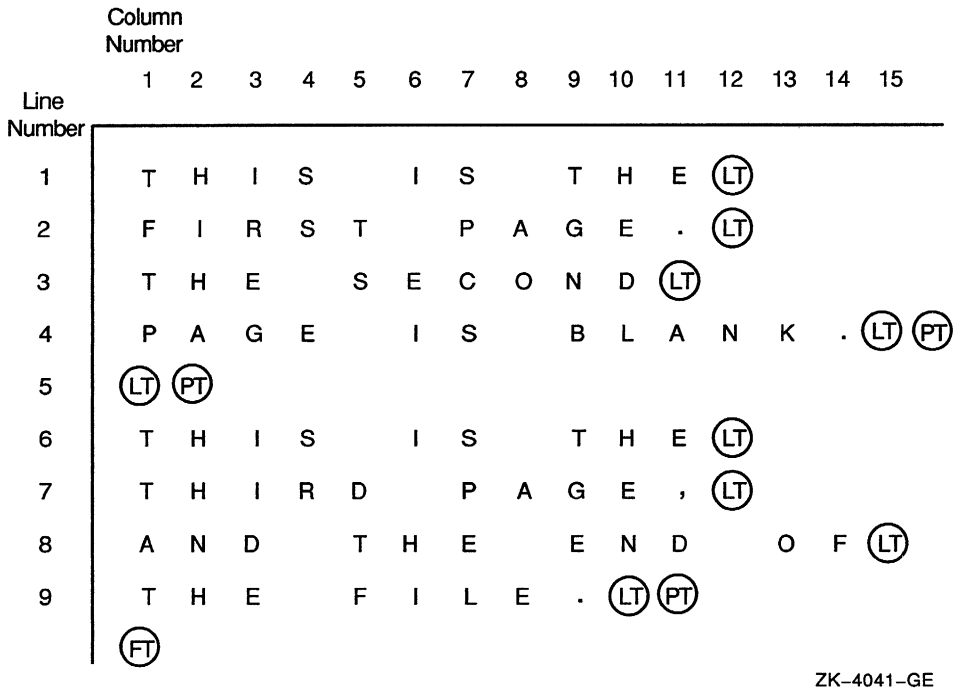
Example 3–11 (Cont.): Example of Flexible TEXT_IO

```
begin
   OPEN(INPUT,IN_FILE,"USER_INPUT:");
   loop
      PUT("Enter a real or integer format number (0 to exit) ");
      exit when NUM = 0.0;
      GET_NUM(INPUT,NUM);
      NEW_LINE;
      PUT("Ok, we received: ");
      PUT(NUM);
      NEW_LINE;
   end loop;
end THIRD_IO_EXAMPLE;
```

3.7.2 Line Terminators, Page Terminators, and File Terminators

The Ada language defines "logical" text files and text file operations in terms of line terminators, page terminators, and file terminators (see Chapter 14 of the *VAX Ada Language Reference Manual*). This definition means that a text file is logically structured so that the end of a line is marked by a line terminator (LT), the end of a page is marked by a line terminator followed by a page terminator (LT PT), and the end of a file is marked by a line terminator followed by a page terminator followed by a file terminator (LT PT FT). Figure 3–3 shows a simple, three-page text file.

Figure 3–3: An Ada Text File, Showing Line, Page, and File Terminators



VAX Ada interprets these terminators is as follows:

- A line terminator (LT) is designated by the end of a VMS RMS record, except when the next record in the file logically represents a line terminator followed by a page terminator (LT PT; see the next item). In an empty file, a line terminator is designated by the end of the file.
- A line terminator followed by a page terminator (LT PT) is designated by one of the following:
  - An entire record consisting of a single form-feed control character (for text files with variable-length records).
  - An entire record with a form-feed control character as the first byte of the record (for text files with fixed-length records).

- An empty record with VMS RMS PRN information that indicates a form-feed control character (for variable-length with fixed-length control records and files created with the CARRIAGE_CONTROL PRINT attributes).
- The end of the file, whenever the last record of the file does not itself represent a page terminator (that is, when the last record does not represent a line terminator followed by a page terminator; LT PT).

- A file terminator (FT) is designated by the end of the file. An empty file thus represents a line terminator followed by a page terminator followed by a file terminator (LT PT FT). If the file is not empty and the last record of the file does not represent a line terminator followed by a page terminator (LT PT) (if, for example, the file consists of a single line ending in only a line terminator), then the end of the file represents a page terminator followed by a file terminator (PT FT). When the last record of the file represents a line terminator followed by a page terminator (LT PT), the end of the file is a file terminator (FT).

For example, an external file created by the following three operations contains exactly one empty record:

```
CREATE (MY_FILE);
NEW_LINE(MY_FILE);
CLOSE (MY_FILE);
```

Because the NEW_LINE procedure uses the default spacing of 1, and because no new pages are created, the NEW_LINE in this example produces one line terminator (LT). In this case, a line terminator is represented by a single, empty VMS RMS record in the corresponding external file. (See the *VAX Ada Language Reference Manual* for a complete description of the NEW_LINE procedure.)

By replacing the NEW_LINE procedure with a NEW_PAGE procedure, you would produce a file with one record consisting of a single form-feed control character. (MY_FILE has variable-length records because it is created using the default attributes provided by TEXT_IO.) By completely eliminating the NEW_LINE operation, you would produce an empty file. All three cases mentioned produce the same logical file consisting of a line terminator followed by a page terminator followed by a file terminator (LT PT FT).

3.7.3 Text Input-Output Buffering

VAX Ada TEXT_IO operations are implemented with VMS RMS input-output operations. Because VMS RMS operations always involve complete records, the transfer of characters between a physical input-output device and a VAX Ada text file is complete only when a line terminator is detected. Therefore, in most cases, as characters are read or written to a VAX Ada text file, they are stored in an internal line buffer until a complete record can be transferred through VMS RMS.

Thus, when you are performing either *terminal* or *nonterminal input*, information from the external file is transferred and processed a line (a VMS RMS record) at a time. Hence, terminal input is not processed until the line is terminated by a carriage return (or other line terminator).

It is possible to provide more information in a line than the current input operation needs. In that case, the remaining characters are kept in a buffer to be processed by subsequent input operations. Each time an operation requires more input from the external file, a new read operation from that file is initiated.

When you are performing either *terminal* or *nonterminal output*, the output is also buffered until a line terminator is encountered. In other words, the output is buffered until a NEW_LINE or a NEW_PAGE (or any other operation that in effect performs a NEW_LINE or a NEW_PAGE, such as PUT_LINE) is executed.

Partial buffering is done when you are performing *terminal output* and you have specified the attributes FDL CARRIAGE_CONTROL CARRIAGE_RETURN or CARRIAGE_CONTROL PRINT in a CREATE or OPEN FORM parameter (see Section 3.3). (PRINT is the default CARRIAGE_CONTROL attribute provided by the package TEXT_IO for external files that are terminals; see Table 3–13.)

Partial buffering means that PUT operations to the terminal output file are buffered until one of the following actions occurs:

- Input is attempted for any other file that is associated with the same terminal device (for example, your program executes a PUT, or a series of PUT operations, followed by a GET).
- Execution of one or more PUT operations causes 1000 or more characters to be written to the buffer.

When one of these actions occurs, the contents of the file buffer is output to your terminal, whether or not the record represented by the buffer is complete. For example, the following program buffers the four characters produced by the PUT operations. Then, when the GET is executed, the program prints the letters "abcd" on the screen as a single line and waits for input.

```
with TEXT_IO; use TEXT_IO;
procedure PRINTCHAR is
   C: CHARACTER;
begin
   PUT('a');
   PUT('b');
   PUT('c');
   PUT('d');
   GET(C);
   PUT(C);
end PRINTCHAR;
```

The contents of any text file buffers (partial or full) are also written to your terminal (flushed) whenever your program image exits (such as when an unhandled exception propagates out of a main program). In this situation, all unclosed files are also closed by an exit handler.

3.7.4 TEXT_IO Carriage Control

The FDL CARRIAGE_CONTROL attribute specifies the carriage-control format for a file. You can also use this attribute to control line buffering for files being written to terminal devices.

As described in Section 3.3, you can specify the CARRIAGE_CONTROL attribute with a FORM parameter as follows:

```
TEXT_IO.CREATE (FILE => file_object_name,
                MODE => OUT_FILE,
                NAME => external_file_name,
                FORM => "RECORD; CARRIAGE_CONTROL value;");

TEXT_IO.OPEN   (FILE => file_object_name,
                MODE => OUT_FILE,
                NAME => external_file_name,
                FORM => "RECORD; CARRIAGE_CONTROL value;");
```

The CARRIAGE_CONTROL attribute is a creation-time attribute (see Section 3.3.2), and you cannot use an OPEN procedure to change what was specified when the file was created.

The possible CARRIAGE_CONTROL values are as follows:

CARRIAGE_RETURN	The default if the device is not a terminal; generally provides the desired behavior for most terminal and nonterminal applications.
PRINT	The default if the device is a terminal and the file mode is OUT_FILE; results in the use of a variable-length with fixed-length control (VFC) record format. The control portion of each record contains carriage-control information that indicates line and page boundaries.
NONE	Useful in applications that need to move the cursor randomly and update the screen. Output to files specified with this option is buffered until an operation that requires a line terminator is executed. Calls to PUT_LINE or NEW_LINE can be used to control when the actual VMS RMS line termination operation occurs.
FORTRAN	Useful for applications that want to use FORTRAN carriage-control characters.

Table 3–14 summarizes the meaning of the FDL CARRIAGE_CONTROL values when they are applied to VAX Ada text files (for both terminal and nonterminal input-output).

Table 3–14: VAX Ada Carriage-Control Options

Option	Kind of Input-Output	Carriage Control
CARRIAGE_RETURN	Terminal input Nonterminal input	Each record corresponds to a single line. A 1-byte record containing a form feed designates a page.
	Terminal output	A VFC record format with a 2-byte control portion is used regardless of what is specified in the form string. The control portion of the record specifies the carriage-control information (line feed, carriage return, null, or page).

(continued on next page)

Table 3–14 (Cont.): VAX Ada Carriage-Control Options

Option	Kind of Input-Output	Carriage Control
	Nonterminal output	The record attributes for the file imply that each record is preceded by a line feed and followed by a carriage return when the file is displayed or printed. A 1-byte record containing a form feed designates a page.
PRINT	Terminal input	Each record corresponds to a single line. A 1-byte record containing a form feed designates a page.
	Nonterminal input	Control information indicates that a page is interpreted as a page terminator. Otherwise, a record is assumed to correspond to a line.
	Terminal output Nonterminal output	A VFC record format with a 2-byte control portion is used regardless of what is specified in the form string. The control portion of the record specifies the carriage-control information (line feed, carriage return, or page).
NONE	Nonterminal input Terminal input	Each record corresponds to a single line. A 1-byte record containing a form feed designates a page.

(continued on next page)

Table 3–14 (Cont.): VAX Ada Carriage-Control Options

Option	Kind of Input-Output	Carriage Control
	Terminal output Nonterminal output	A VMS RMS record is written whenever an operation is executed that requires a line terminator. However, no carriage-control information is written for lines, and the record attributes for the file do not imply that records are preceded by a line feed or followed by a carriage return. A 1-byte record containing a form feed designates a page.
FORTRAN	Terminal input Nonterminal input	The first byte of each record (containing carriage-control information) is considered to be data. Each record corresponds to a single line. A 1-byte record containing a form feed designates a page.
	Terminal output Nonterminal output	No carriage-control information is supplied by VAX Ada. The first byte PUT by the user in each line is interpreted as a FORTRAN carriage-control character (see Table 3–15).

Table 3–15: FORTRAN Carriage-Control Characters

Character	Meaning
' + '	Overprinting: starts output at the beginning of the current line.
' '	Single spacing: starts output at the beginning of the next line.
' 0 '	Double spacing: skips a line before starting output.

(continued on next page)

Table 3–15 (Cont.): FORTRAN Carriage-Control Characters

Character	Meaning
' 1 '	Paging: starts output at the top of a new page.
' $ '	Prompting: starts output at the beginning of the next line and suppresses the carriage return at the end of the line.
ASCII.NUL	Prompting with overprinting: suppresses the line feed at the beginning of the line and the carriage return at the end of the line.

3.7.5 Predefined Instantiations of TEXT_IO Packages

To make your use of the generic TEXT_IO operations more efficient, VAX Ada provides the following predefined library packages that instantiate the integer and floating-point operations for the predefined integer and floating-point types:

Package Name	Instantiation
INTEGER_TEXT_IO	INTEGER_IO(INTEGER)
SHORT_INTEGER_TEXT_IO	INTEGER_IO(SHORT_INTEGER)
SHORT_SHORT_INTEGER_TEXT_IO	INTEGER_IO(SHORT_SHORT_INTEGER)
FLOAT_TEXT_IO	FLOAT_IO(FLOAT)
LONG_FLOAT_TEXT_IO	FLOAT_IO(LONG_FLOAT)
LONG_LONG_FLOAT_TEXT_IO	FLOAT_IO(LONG_LONG_FLOAT)

Thus, instead of writing out the instantiation for INTEGER_IO in each program unit that does text input-output of integers, you can make the predefined package INTEGER_TEXT_IO available to the applicable units (or to your whole program). For example:

```
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure WRITEOUT_INTEGERS is
   A,B: INTEGER;
begin
   A := 10;
   PUT(A);
   B := A**2;
   PUT(B);
    . . .
end WRITEOUT_INTEGERS;
```

Each predefined package is produced by compiling the equivalent of the following instantiation:

```
with TEXT_IO;
package INTEGER_TEXT_IO is new TEXT_IO.INTEGER_IO(INTEGER);
```

If you want to use other TEXT_IO operations, such as string operations or INTEGER_TEXT_IO operations that involve files other than standard files (files that you declare in your program), you must also make the package TEXT_IO available to your program. For example:

```
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure WRITE_STRINGS_AND_INTS is
   X: INTEGER;
   F: FILE_TYPE;
begin
   PUT("The value of X is ");  -- TEXT_IO.PUT
   X := -24;
   PUT(X);                     -- INTEGER_TEXT_IO.PUT
   NEW_LINE;                   -- TEXT_IO.NEW_LINE
   CREATE(F);                  -- TEXT_IO.CREATE
   PUT(F,X);                   -- INTEGER_TEXT_IO.PUT
    . . .
end WRITE_STRINGS_AND_INTS;
```

3.8 Input-Output and Exception Handling

The VAX Ada input-output packages raise errors that are defined in the packages IO_EXCEPTIONS and AUX_IO_EXCEPTIONS. See the *VAX Ada Language Reference Manual* for descriptions of these errors and the operations that raise them. See Chapter 4 of this manual for information on exception handling.

3.9 Input-Output and Tasking

The following sections discuss topics related to tasking and input-output. For more information on tasking and tasking concepts, see Chapter 8.

3.9.1 Synchronization of Input-Output Operations

In VAX Ada, each file operation is synchronized so that a series of operations to the same file takes place sequentially, rather than concurrently. In other words, operations on the same file are "indivisible." Thus, if one task is performing an operation on a file and another task attempts to perform an operation on the same file, VAX Ada causes the second task to wait until the earlier operation is finished. Any number of tasks may be waiting for a file to be released by a task that is performing an operation on that file. This synchronized access allows multiple tasks to perform concurrent input-output operations on the same file without corrupting the file.

For example, if one task executes TEXT_IO.PUT(F,"Reach") and another task concurrently executes TEXT_IO.PUT(F, "Out"), where F is a file object, the external file associated with F receives either "ReachOut" or "OutReach". VAX Ada will not produce "ReOuacth".

If you want to execute concurrent input-output operations from multiple processes, you should use VMS RMS record locking to provide synchronization among the various files.

3.9.2 Task Wait States Caused by Input-Output Operations

In general, VAX Ada input-output operations cause only the executing task to wait until the operation is completed. Other tasks in the process can continue executing while the task executing the input-output operation is waiting.

An exception to this behavior occurs when you perform input-output operations on process-permanent files; for example SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, or SYS$ERROR. (See Section 3.2.2 and the *VMS DCL Concepts Manual* for more information on process-permanent files.) An input-output operation to a process-permanent file will cause most tasks in your program to wait until the input-output operation on the file is completed.

Unlike the read and write operations in the other input-output packages, the GET and PUT procedures in the package TEXT_IO operate on default input and output files if you do not specify a file parameter in the procedure call. In VAX Ada, the default input and output files are represented by the logical names ADA$INPUT and ADA$OUTPUT. If you do not define these logical names, the default input and output files are represented by the default system input and output logical names SYS$INPUT and SYS$OUTPUT.

SYS$INPUT and SYS$OUTPUT are process-permanent files. Thus, if you define ADA$INPUT and ADA$OUTPUT to refer to files that are not process-permanent files, your TEXT_IO GET and PUT operations will cause only the executing task to wait until the operation is completed. Conversely, if you do not define ADA$INPUT and ADA$OUTPUT to refer to files that are not process-permanent files, your TEXT_IO GET and PUT operations will cause most tasks in your process to wait until the operation is completed.

For example, the following commands associate ADA$INPUT and ADA$OUTPUT with TT:

```
$ DEFINE ADA$INPUT TT
$ DEFINE ADA$OUTPUT TT
```

TT is a VMS default logical name that represents the terminal associated with your process (see Table 3–1); it is not a process-permanent file. Chapter 8 has an example program that uses tasks to sort an array of integers while performing terminal input-output; the program shows a situation where the association of ADA$INPUT with TT allows the program to run properly.

If your program does perform input-output operations on process-permanent files, the wait for completion of an input-output operation may be interrupted if an asynchronous system trap (AST) is delivered to the VAX Ada run-time library. For example, an AST may be delivered when the time in a delay statement has expired or when time slicing is in effect. When the wait state is interrupted, tasks of higher priority than the task executing the input-output operation will be allowed to execute. If time slicing is in effect, tasks of equal priority will also be allowed to execute. See Chapter 8 for more information on delay statements, time slicing, task scheduling, and ASTs in tasking programs.

Chapter 4

Exception Handling

VAX Ada exception handling, as defined in Chapter 11 of the *VAX Ada Language Reference Manual*, is implemented using the routines and related VMS system services that comprise the VAX Condition Handling Facility (CHF). However, VAX Ada exception handling is implemented so that you do not need to call CHF routines and services directly. This chapter outlines how VAX Ada exception handling is related to VAX condition handling, and explains how to do exception handling in an Ada program that calls or is called from the external environment.

Ada exception handling and the rules for using the VAX Ada pragmas to import and export exceptions are covered in Chapters 11 and 13 of the *VAX Ada Language Reference Manual*. The CHF is described in the *VMS Run-Time Library Routines Volume*. You should be familiar with the material in these manuals before using the information in this chapter.

4.1 Relationship Between Ada Exception Handling and VAX Condition Handling

All VAX Ada exceptions are encoded as VAX condition values as follows:

- Each predefined exception is encoded as a unique 32-bit condition value.
- Each user-defined exception is encoded either as a unique 32-bit condition value or as the general VAX Ada condition value denoted by the name ADA$_EXCEPTION plus a signal argument that is the address of the exception name.

Section 4.1.1 lists the predefined exceptions and explains the encoding and naming of exceptions in more detail.

Once defined, an exception can be raised. Raising generally causes the CHF procedure LIB$STOP to be called, and the exception's condition value to be passed as one of the signal arguments. A vector of signal arguments and a vector of mechanism arguments are built, and a search is then made for an exception handler. The same sequence of events also occurs if a signaled VAX condition is propagated to an Ada program from the external environment.

In VAX Ada, a general condition handler is automatically established for all stack frames that have exception handlers, and a run-time table of active exception parts is maintained for each frame. (Because blocks generally do not have their own stack frames, this condition handler is established for the subprogram, package body, or task body that contains one or more blocks with exception handlers.) The general condition handler determines which specific Ada exception handler in the frame eventually gains control (if any).

Each frame on the stack is searched for a handler. When a handler is found, the stack is unwound to the handler, and execution continues from there.

If no handler is found, and the exception propagates as far as it can go—to the level of a task or a main program—a VMS or VAX Ada run-time catch-all handler gains control. Catch-all handlers are located in the frames enclosing the main program and library packages, each task body, and each accept body. The catch-all handler produces a message and program execution proceeds as follows:

- If an Ada exception or a VAX condition with a severity of severe reaches an Ada run-time library catch-all handler, the handler displays the exception or condition message, and then the task, main program, or rendezvous becomes completed. (However, when an exception or severe condition leaves an accept body, the message is not displayed because the exception or condition will propagate to both of the tasks involved in the rendezvous.)
- If an unhandled VAX condition (not an Ada exception) with a severity of success, information, warning, or error (any severity except severe) reaches an Ada run-time library catch-all handler, the handler displays the condition message and continues program execution. This behavior is consistent with the behavior of VMS catch-all handlers.
- The Ada run-time library catch-all handlers display a warning when an unhandled exception may have to wait for dependent tasks to terminate.

Catch-all handler messages are sent to the output files denoted by the logical names SYS$OUTPUT and SYS$ERROR (see Chapter 3 for more information on how these names are interpreted). See the *Introduction to the VMS Run-Time Library* for more information about VMS default handlers. See Section 4.5 for more information on exception handling and tasks.

Table 4–1 summarizes the VAX Ada implementation of exception handling.

Table 4–1: Relationship Between Ada Exception Handling and the CHF

Ada Exception Handling	CHF Implementation
Enter an Ada frame with an exception part.[1]	Establish the general VAX Ada condition handler for the surrounding stack frame; if the general handler is already established, maintain information about currently active exception parts with a pointer to an internal VAX Ada run-time table.
Raise an exception.	Signal a condition with a call to LIB$STOP, or signal a hardware-generated condition.
Invoke an exception handler.	Unwind (SYS$UNWIND) to the stack frame of the Ada frame containing the exception part, and to the PC at the start of the appropriate exception handler.
Re-raise the same exception.	Call LIB$STOP with a copy of the signal arguments that caused invocation of the currently active exception part. Note that the SS$_RESIGNAL feature of the CHF is not used to re-raise an exception.
No handler for the exception.	Signal a condition with a call to LIB$SIGNAL (which may result in program continuation at the point after the signal).

[1]The term *Ada frame* refers to a frame, as defined by the Ada language: a block statement or the body of a subprogram, package, task unit, or generic unit. The term *stack frame* (synonymous with the term *call frame*) refers to a run-time VAX structure that stores information about a subprogram, package, task, or instantiated generic unit, and includes information about any contained blocks.

(continued on next page)

Table 4–1 (Cont.): Relationship Between Ada Exception Handling and the CHF

Ada Exception Handling	CHF Implementation
Raise an Ada format exception.[2]	Call LIB$STOP with the condition value ADA$_EXCEPTION and one signal argument. The signal argument is the address of a counted ASCII string (ASCIC string) that is the text of the name of the exception.
Raise a VMS format exception.[2]	Call LIB$STOP with a unique 32-bit VAX condition value.

[2]Exceptions with an Ada format are any user-defined exceptions declared without the import-export pragmas, or any user-defined exceptions declared with the import-export pragmas that specify a value of ADA for the pragma FORM parameter. Exceptions with VMS format are any predefined exceptions or any user-defined exceptions declared with import-export pragmas that specify a value of VMS for the pragma FORM parameter. See Sections 4.1.1, 4.4.1, and 4.4.2.

Note from Table 4–1 that the raising of an exception in an Ada program involves calling the CHF routine LIB$STOP. This action implements the Ada language requirement that the occurrence of an exception must terminate the current Ada frame and transfer control to an exception handler. The effect is that once an exception is raised in an Ada program, control cannot return to the point at which the exception occurred: execution is noncontinuable. See Sections 4.4.4 and 4.4.5 for a discussion of the consequences in mixed-language programs.

In some cases, the exception's signal argument vector may be copied before control is transferred to a handler. For example, if the handler re-raises the exception, the signal argument vector must be copied so that the same signal arguments can be used to raise the exception again. A copy is also needed when an exception is raised at the point of a task rendezvous (the language requires that the exception be propagated to both the called and the calling task). Section 4.1.2 describes how signal argument copying is done, and outlines some side effects.

4.1.1 Naming and Encoding Ada Exceptions

VAX Ada provides predefined exceptions in the packages STANDARD, IO_EXCEPTIONS, AUX_IO_EXCEPTIONS, and SYSTEM. Each predefined exception is encoded with a *VMS format*: a unique 32-bit VAX condition value with a symbolic name. The predefined VAX Ada exceptions have symbolic names of the following form:

```
ADA$_exception_name
```

Thus, the exception CONSTRAINT_ERROR (from the package STANDARD) has the symbolic name ADA$_CONSTRAINT_ERROR, the exception DATA_ERROR (from the package IO_EXCEPTIONS) has the symbolic name ADA$_DATA_ERROR, and so on.

The predefined exceptions are listed in Table 4–2; the situations in which they are raised are described in the *VAX Ada Language Reference Manual*.

Table 4–2: Ada Predefined Exceptions

Package	Exceptions
STANDARD	CONSTRAINT_ERROR NUMERIC_ERROR PROGRAM_ERROR STORAGE_ERROR TASKING_ERROR
SYSTEM	NON_ADA_ERROR
IO_EXCEPTIONS	STATUS_ERROR MODE_ERROR NAME_ERROR USE_ERROR DEVICE_ERROR END_ERROR DATA_ERROR LAYOUT_ERROR
AUX_IO_EXCEPTIONS	LOCK_ERROR EXISTENCE_ERROR KEY_ERROR

Ada allows you to declare your own exceptions so that you can anticipate and handle more specific errors than those covered by the predefined exceptions. For example:

```
INVALID_INPUT : exception;
```

This declaration allows you to use the exception name INVALID_INPUT in a raise statement and as an exception choice in an Ada frame.

In general, user-defined exceptions are encoded with an *Ada format*: they all have the same general 32-bit VAX condition value with the symbolic name ADA$_EXCEPTION, plus an additional signal argument that makes each value unique. This signal argument is the address (32-bit) of the counted ASCII string (ASCIC string) that represents the name of the exception. (The first byte of the string contains the number of characters in the exception name; the remaining bytes contain the characters of the exception name.) The string address is assigned at link time and can change each time the program is linked.

You can cause user-defined exceptions to be encoded with a VMS format by using the pragmas IMPORT_EXCEPTION and EXPORT_EXCEPTION. See Section 4.4 for more information.

4.1.2 Copying Exception Signal Arguments

An exception's signal argument vector is copied if the exception is re-raised by its handler or if the exception is raised at the point of a task rendezvous. This copying is done so that essentially the same signal arguments are used when the exception is propagated.

When a signal argument vector is copied, it is marked as such by being chained to one of two special VAX Ada-specific primary condition values:

- ADA$_EXCCOP, which indicates that the copy is complete
- ADA$_EXCCOPLOS, which indicates that the original signal has been modified and some information may have been lost

The chaining causes ADA$_EXCCOP or ADA$_EXCCOPLOS to become the primary condition in the signal argument vector. The condition that originally caused the exception to be raised then becomes the second condition value in the signal argument vector. The principal reason for chaining the VAX Ada-specific primary condition values to the copied signal argument vector is to prevent incorrect handling—such as continuation—of the original condition. Once a condition has been copied, it has an Ada semantic effect, which does not allow continuation.

Information may be lost from the signal argument vector during copying if the VAX Ada run-time library suspects that the vector has an argument that points to a stack area that must be unwound to reach an exception handler. In general, the optional Formatted ASCII Output (FAO) arguments are the only part of the signal argument vector that is likely to point to such a stack

area. When the VAX Ada run-time library suspects that the FAO arguments point to a stack area, it zeroes the arguments.

Information may be lost only for non-Ada conditions with FAO arguments. Information will not be lost in the following cases:

- For Ada exceptions
- For non-Ada conditions that have no FAO arguments
- For hardware conditions
- For RMS conditions
- For VMS system service conditions

Whether or not information has been lost by copying, the handling of an exception in Ada is not affected. The handling of the exception in non-Ada code is affected only if messages that depend on zeroed FAO arguments are involved; such messages are printed with embedded FAO directives (for example, !AS, !UL, and so on).

4.1.3 The Matching of Ada Exceptions and System-Defined VAX Conditions

In VAX Ada, the matching of exceptions to exception choices depends on the matching of the condition values assigned to the exception and the choice names. In particular, two user-defined, Ada format exceptions—exceptions encoded as ADA$_EXCEPTION plus an ASCIC string—match only if the addresses of their ASCIC strings match. If the raised exception is an imported VAX condition (see Section 4.4.1), it matches an exception choice only if the name in the exception choice matches the internal name of the imported VAX condition. Imported VAX conditions also match the exception choice **others** and the exception name SYSTEM.NON_ADA_ERROR (see Section 4.4.3).

Some VAX conditions are treated as being equivalent to certain Ada predefined exceptions. When one of these conditions is signaled during the execution of an Ada program, the effect is as if the predefined exception were raised, and the condition can be caught by an Ada exception handler that exists to catch the predefined exception. Table 4–3 lists the VAX conditions that have Ada predefined equivalents.

Table 4–3: VAX Conditions that Match Ada Exceptions

Condition Name	Meaning	Exception Name
SS$_INTDIV	Integer divide by zero trap	CONSTRAINT_ERROR
SS$_FLTDIV	Floating/decimal divide by zero trap	CONSTRAINT_ERROR
SS$_FLTDIV_F	Floating divide by zero fault	CONSTRAINT_ERROR
SS$_INTOVF	Integer overflow trap	CONSTRAINT_ERROR
SS$_FLTOVF	Floating overflow trap	CONSTRAINT_ERROR
SS$_FLTOVF_F	Floating overflow fault	CONSTRAINT_ERROR

Note that these VAX conditions match CONSTRAINT_ERROR, but are not converted to CONSTRAINT_ERROR. If one of them is signaled and propagates out of a main program, the result is an informational message identifying the event as a CONSTRAINT_ERROR (that is, you could have caught it with a CONSTRAINT_ERROR handler), as well as the error message and traceback associated with the actual condition.

4.2 Making the Best Use of Ada Exception Handling

To make the best use of Ada exception handling, keep these two principles in mind:

- Code handlers for specific exceptions. In particular, do not use a general **others** handler when you could write an explicit handler for a specific exception.
- Allow unexpected exceptions to propagate, instead of being absorbed.

For example, if you use an **others** choice to handle the predefined input-output exception END_ERROR, the handler will also be invoked for any unexpected exception, such as "disk quota exceeded." A better solution would be to provide a specific handler for END_ERROR, and allow unexpected exceptions to propagate, thereby making them visible.

Similarly, the following construct absorbs all exceptions without allowing them to propagate and without issuing a message:

```
when others => null;
```

Such a statement is unlikely to be able to handle all possible exceptions and recover correctly.

To ensure that unexpected exceptions do propagate and become visible, end your **others** exception choices with a raise statement. For example:

```
begin
   -- Sequence of statements for a block.
    . . .
exception
   when SINGULAR | CONSTRAINT_ERROR =>
      PUT (" MATRIX IS SINGULAR ");
   when others =>
      -- Perform some cleanup operations.
       . . .
      raise;
end;
```

Here, if an exception other than SINGULAR or CONSTRAINT_ERROR is raised, the **when others** handler gains control, and the exception will be re-raised in the containing frame.

4.3 Suppressing Checks

In accordance with the language definition, VAX Ada provides a set of run-time checks that underlie the predefined exceptions. The *VAX Ada Language Reference Manual* explains each check that the compiler performs and gives the corresponding exceptions that can arise. Table 4–4 summarizes this correspondence.

Table 4–4: Run-Time Checks and Their Corresponding Predefined Exceptions

Check	Predefined Exception Raised
ACCESS_CHECK DISCRIMINANT_CHECK INDEX_CHECK LENGTH_CHECK RANGE_CHECK	CONSTRAINT_ERROR
DIVISION_CHECK OVERFLOW_CHECK	CONSTRAINT_ERROR
ELABORATION_CHECK	PROGRAM_ERROR
STORAGE_CHECK	STORAGE_ERROR

Example 4–1: Use of Pragma SUPPRESS_ALL

```
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure MAIN is

   procedure A is                    -- Checks are not
   begin                             -- suppressed in A.
       . . .
   end;

   procedure B is separate;          -- Checks are
                                     -- suppressed in B.

   procedure C is separate;          -- Checks are not
                                     -- suppressed in C.

begin
   B;                                -- Exception B
                                     -- propagated to here.

exception
   when NUMERIC_ERROR | CONSTRAINT_ERROR =>
      PUT_LINE("Division by zero -- propagated up!");

end MAIN;

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

separate (MAIN)
procedure B is
   A: INTEGER := 1;
   I: INTEGER := 0;
begin
   A := A/I;                         -- Exception corresponding
                                     -- to DIVISION_CHECK is
                                     -- raised, even though
   PUT(A);                           -- SUPPRESS_ALL is
                                     -- specified.
end B;
pragma SUPPRESS_ALL;                 -- Pragma must follow the
                                     -- unit to which
                                     -- it applies.

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

separate (MAIN)
procedure C is
begin
    . . .
end C;
```

To suppress checks, you can use the pragmas SUPPRESS and SUPPRESS_ALL (see the *VAX Ada Language Reference Manual*), or you can use the /NOCHECK qualifier on the ADA and ACS COMPILE and RECOMPILE commands (see *Developing Ada Programs on VMS Systems*).

NOTE

When you suppress checks, your program may become erroneous. For example, if you suppress checks in a library unit or subunit that contains a number of arrays, you will suppress INDEX_CHECK, but array processing will continue, whether or not you exceed the specified ranges. The results of that unit or subunit will be unpredictable. If you are using the pragmas SUPPRESS or SUPPRESS_ALL of the /NOCHECK qualifier to improve the run-time performance of your program, consider using the techniques for eliminating checks discussed in Chapter 9.

The presence of the pragmas SUPPRESS or SUPPRESS_ALL or the use of the /NOCHECK qualifier does not guarantee that exceptions will not be raised. For example, certain checks are not suppressed in VAX Ada. These checks are the hardware checks DIVISION_CHECK and OVERFLOW_CHECK (for floating-point types), where the hardware catches the error and passes control directly to the operating system. STORAGE_CHECK is also not suppressed (except for stack checks). Thus, an exception may be propagated from a called unit in which the corresponding check was suppressed; the predefined exception corresponding to DIVISION_CHECK is propagated from subunit B to MAIN_EXCEPTIONS in Example 4–1.

4.4 Mixed-Language Exception Handling

VAX Ada provides the pragmas IMPORT_EXCEPTION and EXPORT_EXCEPTION for use in a mixed-language programming environment:

- The pragma IMPORT_EXCEPTION allows you to import an exception declared in a non-Ada program and handle it within your Ada program.
- The pragma EXPORT_EXCEPTION allows you to export an Ada exception so that it can be treated as a VAX condition in a non-Ada program.

VAX Ada also provides specific facilities for signaling and handling VAX conditions in the VMS environment: the function SYSTEM.IMPORT_VALUE, the package CONDITION_HANDLING, and the package STARLET.

The following sections explain how to use these features.

For information on testing status values returned by VMS system routines, see Chapter 6.

4.4.1 Importing Exceptions

The pragma IMPORT_EXCEPTION associates an Ada exception name with either a VAX condition value or another Ada exception name—both external to your program—and then allows you to use that name in your Ada program. The full syntax and usage rules for this pragma are given in Chapter 13 of the *VAX Ada Language Reference Manual*. The syntax is summarized here for convenience:

```
pragma IMPORT_EXCEPTION
    (internal_name [, external_designator]
       [, [FORM => ] ADA | VMS]
       [, [CODE => ] static_integer_expression]);

internal_name ::= [INTERNAL] => simple_name

simple_name ::= identifier

external_designator ::= [EXTERNAL] => external_symbol

external_symbol ::= identifier | string_literal
```

You can use the pragma IMPORT_EXCEPTION to associate an Ada exception name with either a numeric condition value or a global symbol that denotes a VAX condition: the CODE parameter represents a numeric condition value, and the external designator represents a global symbol. You can specify the format of the exception with the FORM parameter (see Section 4.1.1 for definitions of Ada and VMS formats; the format for imported exceptions is VMS by default). You can raise an imported exception with a raise statement and handle it with an Ada exception handler that names the exception.

In the following example, the procedure QUADRATIC_FORMULA computes and prints the real roots of a quadratic equation. The procedure imports the VAX condition MTH$_SQUROONEG; the pragma IMPORT_EXCEPTION associates the exception name SQRT_NEGATIVE with the VAX condition MTH$_SQUROONEG. If the call to the SQRT function in the procedure QUADRATIC_FORMULA attempts to compute the square root of a negative number, the exception SQRT_NEGATIVE is raised. Control then transfers to the exception handler, which completes execution of the procedure by printing a message.

```
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
with FLOAT_MATH_LIB; use FLOAT_MATH_LIB;
   -- These packages are predefined by VAX Ada
   -- for convenience.
with TEXT_IO; use TEXT_IO;
procedure QUADRATIC_FORMULA is

   A, B, C, D : FLOAT;
   SQRT_NEGATIVE : exception;
      pragma IMPORT_EXCEPTION (
         SQRT_NEGATIVE,       -- Internal_name.
         "MTH$_SQUROONEG");   -- External_designator.
                              -- By default use
                              -- VMS format.

begin

   GET(A); GET (B); GET(C);
   D := SQRT(B**2 - 4.0*A*C);-- Exception will be
                              -- raised here if
                              -- B**2 - 4.0*A*C
                              -- is negative.
   PUT("Real Roots : X1 = ");
   PUT((-B - D)/(2.0*A));
   PUT(" X2 = ");
   PUT((-B + D)/(2.0*A));

exception
   when SQRT_NEGATIVE =>
      PUT_LINE("Imaginary Roots.");

end QUADRATIC_FORMULA;
```

The next example is a declaration that shows how you can import the VAX condition SS$_ACCVIO using its numeric code. No external symbol is referenced in this case (it is illegal if the code option is specified). Because it is the default, the VMS format will be used (the VMS format is the required format for importing VAX conditions).

```
SS_ACCVIO : exception;
pragma IMPORT_EXCEPTION (
   SS_ACCVIO,         -- Internal_name.
   CODE => 16#0C#);   -- Numeric condition value.
```

You can give a VAX condition value to an Ada exception by first defining a message symbol condition with the VAX Message Utility (see the *VMS Message Utility Manual*), and then using IMPORT_EXCEPTION to associate the Ada exception name with the message symbol. For example:

```
NEW_VMS : exception;
pragma IMPORT_EXCEPTION (
   NEW_VMS,              -- Internal_name.
   "MSG$_ERRORS",        -- External_designator of a message symbol
                         -- defined with the VAX Message Utility.
   FORM => VMS);         -- Explicitly specify VMS format.
```

Sections 4.4.4 and 4.4.5 provide additional discussion and examples of importing and handling VAX conditions in the VMS environment.

4.4.2 Exporting Exceptions

The pragma EXPORT_EXCEPTION associates an Ada exception with either a VAX condition value or another Ada exception name, and then allows you to use that name in the external environment. The full syntax and usage rules for this pragma are given in Chapter 13 of the *VAX Ada Language Reference Manual*. The syntax is summarized here for convenience:

```
pragma EXPORT_EXCEPTION
  (internal_name [, external_designator]
     [, [FORM => ] ADA | VMS ]
     [, [CODE => ] static_integer_expression]);

internal_name ::= [INTERNAL] => simple_name

simple_name ::= identifier

external_designator ::= [EXTERNAL] => external_symbol

external_symbol ::= identifier | string_literal
```

If you export a VMS format exception (see Section 4.1.1) to a non-Ada routine, that routine can treat the exception as an ordinary VAX condition. In other words, the routine can process the exception using its own condition-handling mechanisms (for example, with ON units if the routine is written in PL/I, or with LIB$ESTABLISH if the routine is written in FORTRAN).

If you export an Ada format exception (see Section 4.1.1), the external designator becomes a global symbol, which is the address of the exception's ASCIC string name. Because an exception with the Ada format is unique only in the address of its ASCIC string, any non-Ada routine to which such an exception is exported must examine the exception's signal arguments to determine a match. Similarly, any non-Ada routine to which an Ada format exception is propagated must determine if the primary condition is ADA$_EXCEPTION, and, if so, must examine the exception's first FAO signal argument to determine if the argument matches the value of the external designator.

The following examples show how to declare Ada and VMS format exceptions, so that they can be exported to the external environment:

```
ADA_ERROR : exception;
pragma EXPORT_EXCEPTION
   (ADA_ERROR,                -- Internal_name.
    "MY_PACKAGE_ADA",         -- External_designator.
    FORM => ADA);             -- Ada format.

VMS_ERROR : exception;
pragma EXPORT_EXCEPTION
   (VMS_ERROR,                -- Internal_name.
    "MY_PACKAGE_ADA",         -- External_designator.
    FORM => VMS,              -- VMS format.
    CODE => 16#8018004#);     -- VAX condition value.
```

4.4.3 The Exception Choice NON_ADA_ERROR

To allow you to treat non-Ada conditions as a special subclass of Ada exceptions, VAX Ada provides the exception choice NON_ADA_ERROR in the package SYSTEM. NON_ADA_ERROR matches itself and any VAX condition whose facility field is not ADA. It is encoded as a predefined exception with the unique condition value ADA$_NON_ADA_ERROR.

NON_ADA_ERROR also matches Ada exceptions for which the pragma IMPORT_EXCEPTION or EXPORT_EXCEPTION is given and for which the VMS format has been specified.

4.4.4 Signaling VAX Conditions

VAX Ada provides two ways to signal a VAX condition from an Ada program:

- Import the condition using the pragma IMPORT_EXCEPTION and use an Ada raise statement to raise the imported exception.
- Signal the condition directly, using a form of the VMS Run-Time Library routine LIB$SIGNAL or LIB$STOP. The VAX Ada package CONDITION_HANDLING provides the procedures SIGNAL and STOP for this purpose.

When you import a condition and signal it with an Ada raise statement, the condition behaves like an Ada exception, according to Ada semantics. Consequently, you can handle it with an Ada exception handler, but regardless of the condition's severity, the signal is noncontinuable.

For example, the Ada subprogram in Example 4–2 calls the system service SYS$GETJPIW, which returns information about one or more VMS processes. When the condition SS$_NONEXPR occurs in Example 4–2, SYS$GETJPIW returns the appropriate warning status. However, because SS$_NONEXPR is imported and treated as an Ada exception, its severity becomes severe. Because an Ada exception handler exists for the exception SS_NONEXPR, control passes to the handler.

Example 4–2: Handling SYS$GETJPIW Status Values as Ada Exceptions

```
with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with STARLET; use STARLET;
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure GETJPI_ADA is

    -- Declare the variables needed to make the call and print
    -- some results.
    --
    JPI_STATUS    : COND_VALUE_TYPE;
    PID_ADDRESS   : UNSIGNED_LONGWORD := 2;
    PIDADR        : ADDRESS := PID_ADDRESS'ADDRESS;
    IOSB_VALUE    : IOSB_TYPE;
    ACCOUNT       : STRING(1..8);
    DFPFC         : INTEGER;
    FREP0VA       : INTEGER;
    FREP1VA       : INTEGER;
    FREPTECNT     : INTEGER;
    GPGCNT        : INTEGER;
     . . .

    JPI_ITEM_LIST : constant ITEM_LIST_TYPE :=
      ((8, JPI_ACCOUNT,    ACCOUNT'ADDRESS,    ADDRESS_ZERO),
       (4, JPI_DFPFC,      DFPFC'ADDRESS,      ADDRESS_ZERO),
       (4, JPI_FREP0VA,    FREP0VA'ADDRESS,    ADDRESS_ZERO),
       (4, JPI_FREP1VA,    FREP1VA'ADDRESS,    ADDRESS_ZERO),
       (4, JPI_FREPTECNT, FREPTECNT'ADDRESS, ADDRESS_ZERO),
       (4, JPI_GPGCNT,     GPGCNT'ADDRESS,     ADDRESS_ZERO),
        . . .
       (0, 0, ADDRESS_ZERO, ADDRESS_ZERO));
```

(continued on next page)

Example 4–2 (Cont.): Handling SYS$GETJPIW Status Values as Ada Exceptions

```
   -- Declare the possible errors as Ada exceptions.
   --
    . . .
   NONEXPR: exception;
   pragma IMPORT_EXCEPTION(NONEXPR,"SS$_NONEXPR");
   NOPRIV : exception;
   pragma IMPORT_EXCEPTION(NOPRIV, "SS$_NOPRIV");
    . . .
   -- Print out the values returned in the item list.
   --
   procedure PRINT_RESULTS is
   begin
      PUT_LINE("Account    = " & ACCOUNT);

      PUT("Default page fault cluster size =");
      PUT(DFPFC);
      NEW_LINE;

      PUT_LINE("First free program region");
      PUT("page address (P0)                =");
      PUT(FREP0VA);
      NEW_LINE;
       . . .
   end PRINT_RESULTS;
begin
   -- Call SYS$GETJPIW using the interface from the package STARLET.
   --
   GETJPIW(STATUS => JPI_STATUS,     PIDADR => PIDADR,
           ITMLST => JPI_ITEM_LIST, IOSB   => IOSB_VALUE);

   -- Check the result status; raise exceptions if the result is
   -- not normal.
   --
   if JPI_STATUS = SS_NORMAL then
      PRINT_RESULTS;
   else
      if JPI_STATUS = SS_NONEXPR then
         raise NONEXPR;
      end if;
       . . .
   end if;
```

(continued on next page)

Example 4–2 (Cont.): Handling SYS$GETJPIW Status Values as Ada Exceptions

```
-- Handle the exceptions.
--
exception
    . . .
   when NONEXPR =>
      PUT_LINE("Nonexistent process");
   when NOPRIV  =>
      PUT_LINE("Insufficient privileges");
    . . .
end GETJPI_ADA;
```

When you signal a condition using the CONDITION_HANDLING.SIGNAL procedure (or another Ada equivalent to the VMS Run-Time Library LIB$SIGNAL routine), the condition behaves like a VAX condition, according to CHF rules. If the condition's severity is not severe (error, warning, or informational), then the signal is continuable. Example 4–3 rewrites Example 4–2 to achieve this effect. In Example 4–3, CONDITION_HANDLING.SIGNAL is used so that the warning status of the NONEXPR condition is preserved and continuation occurs.

Example 4–3: Handling SYS$GETJPIW Status Values as VMS Conditions

```
with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with STARLET; use STARLET;
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure GETJPI_VMS is

    -- Declare the variables needed to make the call and print
    -- some results.
    --
    JPI_STATUS    : COND_VALUE_TYPE;
    PID_ADDRESS   : UNSIGNED_LONGWORD := 2;
    PIDADR        : ADDRESS := PID_ADDRESS'ADDRESS;
    IOSB_VALUE    : IOSB_TYPE;
    ACCOUNT       : STRING(1..8);
    DFPFC         : INTEGER;
    FREP0VA       : INTEGER;
    FREP1VA       : INTEGER;
    FREPTECNT     : INTEGER;
    GPGCNT        : INTEGER;
     . . .

    JPI_ITEM_LIST : constant ITEM_LIST_TYPE :=
      ((8, JPI_ACCOUNT,    ACCOUNT'ADDRESS,    ADDRESS_ZERO),
       (4, JPI_DFPFC,      DFPFC'ADDRESS,      ADDRESS_ZERO),
       (4, JPI_FREP0VA,    FREP0VA'ADDRESS,    ADDRESS_ZERO),
       (4, JPI_FREP1VA,    FREP1VA'ADDRESS,    ADDRESS_ZERO),
       (4, JPI_FREPTECNT, FREPTECNT'ADDRESS, ADDRESS_ZERO),
       (4, JPI_GPGCNT,     GPGCNT'ADDRESS,     ADDRESS_ZERO),
        . . .
       (0, 0, ADDRESS_ZERO, ADDRESS_ZERO));

   -- Print out the values returned in the item list.
   --
   procedure PRINT_RESULTS is
   begin
      PUT_LINE("Account    = " & ACCOUNT);

      PUT("Default page fault cluster size =");
      PUT(DFPFC);
      NEW_LINE;

      PUT_LINE("First free program region");
      PUT("page address (P0)                =");
      PUT(FREP0VA);
      NEW_LINE;
       . . .
   end PRINT_RESULTS;
```

(continued on next page)

Example 4–3 (Cont.): Handling SYS$GETJPIW Status Values as VMS Conditions

```
begin
   -- Call SYS$GETJPIW using the interface from the package STARLET.
   --
   GETJPIW(STATUS => JPI_STATUS,    PIDADR => PIDADR,
           ITMLST => JPI_ITEM_LIST, IOSB   => IOSB_VALUE);

   -- Check the result status; signal if the result is not normal.
   --
   if JPI_STATUS = SS_NORMAL then
      PRINT_RESULTS;
   else
      --
      -- SS_NONEXPR has a status of warning; after it is signaled,
      -- execution can continue. In this case, change the PID value
      -- and call GETJPIW again, so that information about the
      -- current process is returned.
      --
      if JPI_STATUS = SS_NONEXPR then
         SIGNAL(SS_NONEXPR);
         PIDADR := ADDRESS_ZERO;
         GETJPIW(STATUS => JPI_STATUS,    PIDADR => PIDADR,
                 ITMLST => JPI_ITEM_LIST, IOSB   => IOSB_VALUE);
         PRINT_RESULTS;
      end if;
       . . .
   end if;

end GETJPI_VMS;
```

If, in Example 4–3, you were to use the CONDITION_HANDLING.STOP procedure (or another Ada equivalent to the VMS Run-Time Library LIB$STOP routine), the effect would be identical to the effect of an Ada raise statement. Continuation would not occur.

In any case, when working in a mixed-language environment with Ada subprograms, do not depend on the severity of a particular status value. Any Ada exception or VAX condition that is raised becomes severe, and is beyond the control of the code that originally signaled it.

4.4.5 Effects of Handling VAX Conditions from an Ada Program

If an Ada subprogram handles a VAX condition that is signaled and/or propagated from an external routine, the handler causes the signal to behave according to Ada semantics. This rule affects the use of certain VAX Run-Time Library fault handlers in a program that calls Ada subprograms (see Section 4.4.6), and it has the following consequences in any mixed-language program:

- An Ada subprogram that does not contain any exception handlers (which name the raised exception) is transparent to the CHF when a VAX condition is signaled.
- A VAX condition that is handled by an Ada handler is converted to a noncontinuable exception.

Thus, when programming in more than one language, be careful about using general or global condition handlers. In the Ada portions of your program, use the following handling mechanisms carefully, because all of them catch VAX conditions:

- The exception choice **others**—catches any VAX condition or Ada exception that does not have an explicit handler
- The VAX Ada predefined exception SYSTEM.NON_ADA_ERROR—catches all VAX conditions (see Section 4.4.3)
- Predefined Ada exceptions that match VAX conditions (see Section 4.1.3)

NOTE

The exception choice **others** does not catch the following VAX conditions:

SS$_DEBUG
SS$_UNWIND

These two VAX conditions are used to implement the VAX Ada run-time environment, and are never caught by an Ada exception handler. Even if you import one of these conditions and name it in an Ada exception handler, the handler is never invoked to respond to the condition.

For each subprogram that you write, anticipate the errors that can occur, and explicitly name each possible error in an exception handler. In addition, make sure you understand the behavior of a subprogram (and the main program) if you are adding it to an existing application.

For example, consider the following situation:

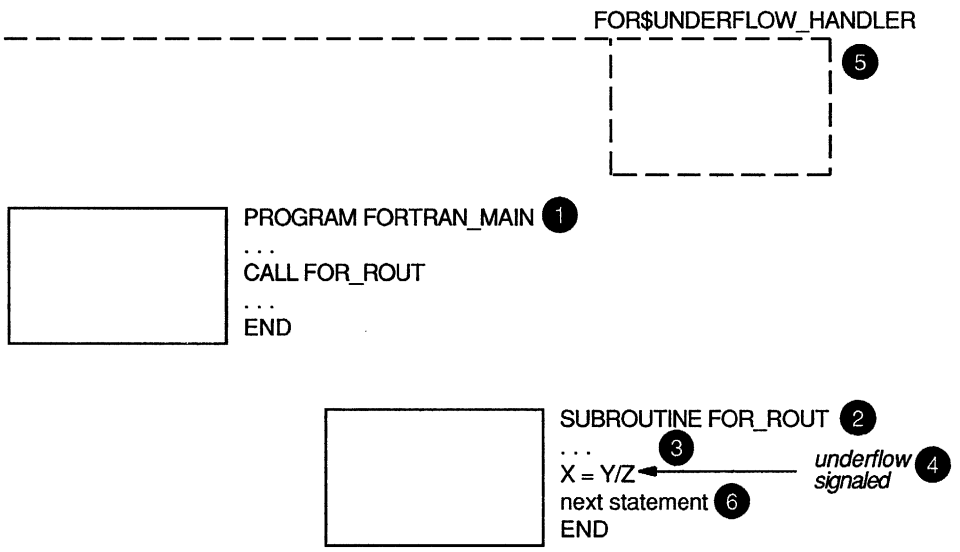
- A FORTRAN program P1_FOR calls an Ada subprogram P2_ADA.
- The Ada subprogram calls another FORTRAN program, P3_FOR.

In this situation, P3_FOR could signal a condition, and P1_FOR could handle the condition and continue the execution of P3_FOR. Because it did not handle the condition, P2_ADA would be unaffected. From the Ada subprogram's point of view, the condition would never have happened.

Alternatively, consider the following situation. You have a working FORTRAN program, called FORTRAN_MAIN, for which you have enabled underflow conditions at compile time (see the *VAX FORTRAN User's Guide*). A default FORTRAN (FOR$UNDERFLOW_HANDLER) is established at the level of the main program to process any underflow conditions that arise. In other words, when an operation in the program underflows and the condition is not processed by another handler, FOR$UNDERFLOW_HANDLER assumes control. This handler keeps a count of the number of underflow conditions and continues execution from the point of the signal.

FORTRAN_MAIN calls several routines, including a FORTRAN subroutine named FOR_ROUT. Figure 4–1 shows the call frames for these routines; the circled numbers indicate the order of execution. If an underflow condition is signaled in FOR_ROUT, FOR$UNDERFLOW_HANDLER gains control, processes the condition, and continues execution of FOR_ROUT.

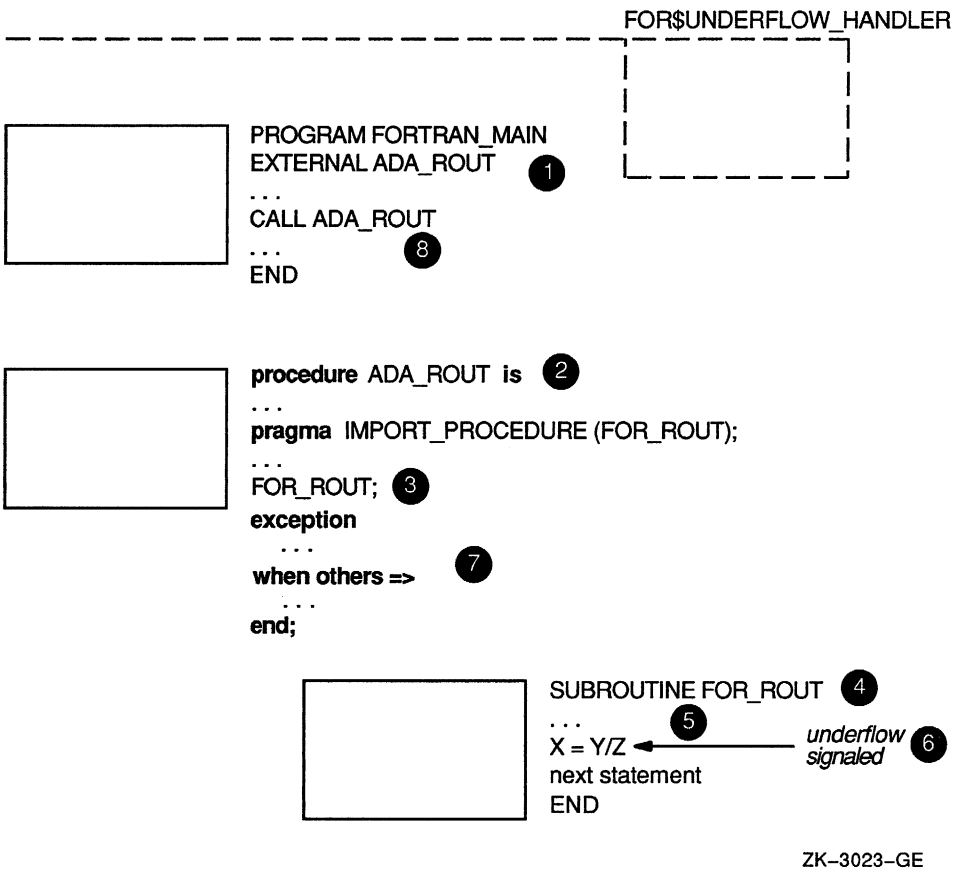
Figure 4–1: Execution of a FORTRAN Program with FOR$UNDERFLOW_HANDLER



Now suppose that when enhancing the program, you add a call to an Ada subprogram called ADA_ROUT, which contains an exception handler with the exception choice **others**. You set the program up so that FORTRAN_MAIN calls ADA_ROUT, and ADA_ROUT calls FOR_ROUT. Figure 4–2 shows the calling sequence for this series of routines. If FOR_ROUT signals underflow, the handler in ADA_ROUT gains control and converts the condition to a noncontinuable exception.

If the handler in ADA_ROUT does not re-raise the exception, execution resumes in FORTRAN_MAIN at the statement after the call to ADA_ROUT. FOR_ROUT does not continue executing, which was the original intent. If the Ada handler re-raises the exception, the exception is propagated to FORTRAN_MAIN and FOR$UNDERFLOW_HANDLER. When FOR$UNDERFLOW_HANDLER attempts to continue from a noncontinuable exception, the result is a fatal error and execution terminates.

Figure 4–2: The Effect of an Ada Procedure Containing an Others Handler



There are two flaws in this example:

- The global error handler, which causes the underflow condition to propagate through the Ada subprogram, instead of remaining local to FOR_ROUT
- The **others** choice in the Ada subprogram, which catches a VAX condition it never intended to catch

One solution for improving this example is to use VMS Run-Time Library calls to explicitly establish FOR$UNDERFLOW_HANDLER in the places where you want to handle underflow. Figure 4–3 shows a repaired version of the FORTRAN program, which instead establishes FOR$UNDERFLOW_HANDLER for the subroutine FOR_ROUT. Then, when underflow is signaled in FOR_ROUT, the handler gains control, processes the condition, and continues execution of FOR_ROUT. The signal is never propagated to the Ada procedure ADA_ROUT.

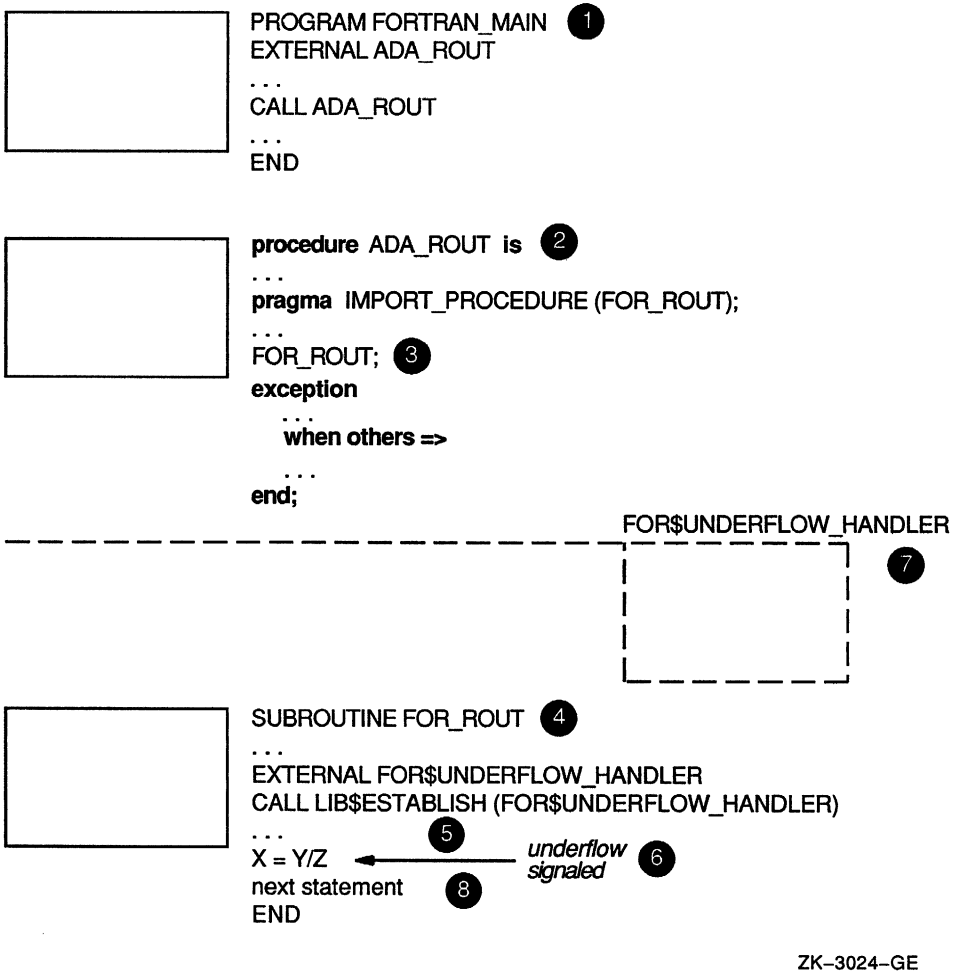
4.4.6 Fault Handlers

Fault handlers are usually established as catch-all stack handlers, or they are called from condition handlers (such as the VMS Run-Time Library routine LIB$DECODE_FAULT); they need the signal information generated at the time the fault occurred in order to execute properly. Because VAX Ada exception handling involves unwinding the stack frame to the start of the handler, signal information is lost when an exception is handled in Ada (the saved signal arguments described in Section 4.1.2 are accessible only to the Ada run-time library and not to the underlying CHF routines). Thus, you cannot do the following:

- Establish a fault handler in Ada.
- Call a VMS Run-Time Library fault handler from an Ada exception part (even if the handler is imported with the pragma IMPORT_PROCEDURE or using the package STARLET).
- Catch a fault in an Ada program when that program has called a procedure in another language, and then continue from the point of the fault.

The only way to set up a fault handler in Ada is to use the VMS vectored exception mechanism. In other words, you must call the system service SYS$SETEXV to assign the address of a fault handler to the primary or secondary exception vector. Because the CHF looks for a handler in the primary and secondary exception vectors before it looks for a handler in an Ada frame, the vectored fault handler gains control before the search for an Ada handler can begin. The fault handler then processes the fault, and the fault is dismissed before it can ever be propagated to an Ada handler.

Figure 4–3: FOR$UNDERFLOW_HANDLER Established for a FORTRAN Subroutine



See Chapter 6 of this manual for information on how to call system services, and see the *Introduction to the VMS Run-Time Library* for an explanation of exception vectors.

4.5 Exceptions and Tasking

Exception handling in VAX Ada interacts with tasking in several ways.

First, the Ada language requires that an unhandled exception terminate the task in which it is raised or to which it is propagated. When this situation occurs in a VAX Ada program, the VAX Ada run-time library displays the exception message to warn you that the task is terminating. You can also use the VMS Debugger to diagnose this situation (see *Developing Ada Programs on VMS Systems*).

NOTE

If you do not want your software to produce task termination messages, you should include exception handlers in those task bodies to which you expect unhandled exceptions to propagate. For example, if you expect that the user-defined exception ALL_DONE will normally cause task termination messages in one of your tasks, you should include the following code (or its equivalent) in the exception part of the affected task body:

```
when ALL_DONE => null;
```

The handler absorbs the exception and prevents it from propagating further. The handler also allows a reader of your code to infer that the termination resulting from this exception is to be expected.

Second, as required by the Ada language, the propagation of an exception from an Ada frame must await the termination of all tasks that depend on that frame. If a dependent task never terminates, then the exception is never propagated. You can avoid this situation as follows:

- By coding an exception handler to call an entry in the task that causes the task to terminate
- By making sure that the task will eventually reach a select statement with an open terminate alternative
- By using an abort statement (use of the abort statement is not recommended except as a last resort; see Section 8.4.5 for a more complete discussion)

To help you diagnose this situation, the task termination messages displayed by the Ada run-time library appear when waiting begins for dependent tasks. You can also use the VMS Debugger to diagnose situations in which an exception may wait forever for a dependent task (see *Developing Ada Programs on VMS Systems*).

Third, VAX Ada requires that tasks declared in a library package or defined by an access type declared in a library package be terminated before execution is returned to the VMS DCL command interpreter. This additional VAX Ada requirement (which is formally supported by Ada language interpretation AI-00399) means that such tasks are guaranteed to reach one of the language-defined points at which task termination can occur. A consequence of this rule is that propagation of an exception out of the frame of the main program does not occur until all tasks declared in library packages terminate. After all such tasks have terminated, the exception is propagated to a VMS default handler (see Section 4.1). In cases where such a task fails to terminate, you can use the VMS Debugger to diagnose the problem (see *Developing Ada Programs on VMS Systems*).

Fourth, the Ada language requires that exceptions propagating from an accept statement propagate in the calling task as well as in the called task. To implement this requirement, VAX Ada copies the signal arguments from the called task to the calling task. This copy operation is identical to the copying that occurs when an exception is going to be re-raised (see Section 4.1.2). In particular, the copied exception has a different primary condition, either ADA$_EXCCOP or ADA$_EXCCOPLOS, and some information in the signal arguments may have been lost (zeroed).

Chapter 5

Mixed-Language Programming

All native-mode VAX languages, including VAX Ada, conform to a set of conventions—the VAX Procedure Calling and Condition Handling Standard—that determine how routines are entered and exited, how parameters are passed, and how function results are returned. By conforming to this standard, VAX Ada allows calls from Ada subprograms to external routines and vice versa (external routines are routines that are not part of the Ada program; for example, they can be written in another language).

This chapter explains how to accomplish mixed-language programming with VAX Ada. The full text of the VAX Procedure Calling and Condition Handling Standard is in the VMS manual, *Introduction to VMS System Routines*. See Chapter 13 of the *VAX Ada Language Reference Manual* for the syntax and usage rules for the VAX Ada pragmas that support mixed-language programming; see Chapter 6 of the *VAX Ada Language Reference Manual* for the Ada semantics rules for subprogram calls. You should be familiar with the material in these two manuals before using the information in this chapter.

The calling of system routines and other callable utilities and tools is a specific example of mixed-language programming. See Chapter 6 for more information on that topic.

NOTE

The calling standard uses the term *procedure* to mean any routine entered by a VAX CALL instruction. To avoid confusion with Ada's definition of a procedure, this manual uses the term *routine*.

5.1 Calling External Routines from Ada Subprograms

You call an external routine from an Ada program by first writing an Ada specification for the routine. To indicate that the routine body is not part of the Ada program, you must give the predefined pragma INTERFACE with the Ada specification. You should also give one of the VAX Ada import pragmas (IMPORT_FUNCTION, IMPORT_PROCEDURE, or IMPORT_VALUED_PROCEDURE) and fully specify all of the parameter-passing and function result mechanisms.

The syntax and rules for using the pragma INTERFACE and the VAX Ada import pragmas are described in detail in Chapter 13 of the *VAX Ada Language Reference Manual*. For convenience, the syntax is summarized here (note that the VAX Ada compiler ignores the language name in the pragma INTERFACE):

```
pragma INTERFACE (language_name, subprogram_name);

pragma IMPORT_FUNCTION
   | IMPORT_PROCEDURE | IMPORT_VALUED_PROCEDURE
   (internal_name [, external_designator]
      [, [PARAMETER_TYPES  =>] (parameter_types)]
      [, [RESULT_TYPE      =>] type_mark]
          -- for IMPORT_FUNCTION only
      [, [MECHANISM        =>] mechanism]
      [, [RESULT_MECHANISM =>] mechanism_name]
          -- for IMPORT_FUNCTION only
      [, [FIRST_OPTIONAL_PARAMETER =>] identifier]);

internal_name :: = [INTERNAL =>] simple_name
   | [INTERNAL =>] operator_symbol  -- Can only be used for
                                    -- IMPORT_FUNCTION

external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal

parameter_types ::= null | type_mark {, type_mark}

mechanism ::= mechanism_name
   | (mechanism_name {, mechanism_name})

mechanism name ::= VALUE | REFERENCE
   | DESCRIPTOR [([CLASS =>] class-name)]

class_name ::= UBS | UBSB | UBA | S | SB | A | NCA
```

For example, the following Ada program imports a routine written in VAX Pascal and uses the MECHANISM option to specify the DESCRIPTOR mechanism for its parameter:

```
procedure SIMPLE_IMPORT is
   procedure PRINT_STRING (S : STRING);
   pragma INTERFACE (PASCAL, PRINT_STRING);
   pragma IMPORT_PROCEDURE (
      INTERNAL        => PRINT_STRING,
      PARAMETER_TYPES => (STRING),
      MECHANISM       => (DESCRIPTOR(CLASS=>A)));

   STR: STRING(1..8) := "Surprise";
begin
   PRINT_STRING(STR);
end SIMPLE_IMPORT;
```

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

MODULE Print_String_Module (OUTPUT);

[GLOBAL] PROCEDURE Print_String (S: PACKED ARRAY[L1..V1: INTEGER]
                                    OF CHAR);

   BEGIN
      WRITELN(S);
   END;

END.
```

Once you have coded the Ada specification and the routine to be imported, you compile them using the appropriate compilers (VAX Ada and VAX Pascal, in this case). Then, you link them using the ACS LINK command, and run them using the DCL RUN command. For example:

```
$ ADA SIMPLE_IMPORT
$ PASCAL PRINT_STRING_MODULE
$ ACS LINK SIMPLE_IMPORT PRINT_STRING_MODULE
$ RUN SIMPLE_IMPORT
```

See *Developing Ada Programs on VMS Systems* for more information on linking mixed-language programs.

When calling an external routine, you must ensure that any parameters or function results are passed with the passing mechanism required by the language in which the routine is written. Each VAX language, VMS routine, and callable utility or tool has default conventions that determine how parameters must be passed and function results returned. The VAX language, VMS, and callable utility or tool documentation provides that information.

To determine which mechanisms the VAX Ada compiler has chosen for the parameters in an Ada subprogram, you can use the /WARNING=COMPILATION_NOTES qualifier with any of the Ada compilation commands to determine which mechanisms the compiler has chosen. For example:

```
$ ADA/WARNINGS=COMPILATION_NOTES CALL_FORTRAN

    2       procedure COMPARE (X,Y: INTEGER);
.............................1
%I, (1) Selected/specified passing mechanism is REFERENCE
.............................2
%I, (2) Selected/specified passing mechanism is REFERENCE
```

However, to ensure that the correct passing mechanism is used for each parameter of an external routine, you should explicitly specify each mechanism rather than depending on the compiler-chosen default mechanisms. See Section 5.4 for more information on the default passing mechanisms chosen by the VAX Ada compiler. See Section 5.6 for more information on how to control the passing mechanisms for imported subprogram parameters.

5.2 Calling Ada Subprograms from External Routines

You call an Ada subprogram from a external routine by first writing the specification and body for the Ada subprogram. To indicate that the subprogram will be called from a external routine, you must also give one of the VAX Ada export pragmas (EXPORT_FUNCTION, EXPORT_PROCEDURE, or EXPORT_VALUED_PROCEDURE).

The syntax and rules for using the VAX Ada export pragmas are described in detail in Chapter 13 of the *VAX Ada Language Reference Manual*. For convenience, the syntax is summarized here:

```
pragma EXPORT_FUNCTION
   | EXPORT_PROCEDURE | EXPORT_VALUED_PROCEDURE
   (internal_name [, external_designator]
      [, [PARAMETER_TYPES =>] (parameter_types)]
      [, [RESULT_TYPE     =>] type_mark]);
          -- for EXPORT_FUNCTION only

internal_name :: = [INTERNAL =>] simple_name

external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal

parameter_types ::= null | type_mark {, type_mark}
```

For example, the following VAX Ada subprogram (SWAP) is exported, and then is called from a VAX Pascal main program (Use_swap):

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

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

PROGRAM Use_swap (INPUT,OUTPUT);
Var
   X,Y: INTEGER;
PROCEDURE Swap (VAR Swap1,Swap2: INTEGER);
   EXTERNAL;
Begin
   (* Give X and Y values *)
    . . .
   SWAP(X,Y);
    . . .
End.
```

Alternatively, the procedure SWAP could be exported from an Ada package, as follows:

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

package body SWAP_ROUTINES is
   procedure SWAP (A,B: in out INTEGER) is
      TEMP: INTEGER;
      begin
         TEMP := A;
         A := B;
         B := TEMP;
      end;
    . . .
end SWAP_ROUTINES;
```

When you export an Ada subprogram, you must use the ACS EXPORT command to prepare the subprogram for linking. For example:

```
$ ADA SWAP
$ PASCAL USE_SWAP
$ ACS EXPORT SWAP
$ LINK USE_SWAP,SWAP
```

See *Developing Ada Programs on VMS Systems* for more information on exporting and linking.

When calling an Ada subprogram from an external routine, you must ensure that the parameters are passed in the form required by the Ada subprogram. To determine which mechanisms are required by VAX Ada, use the compiler qualifer /WARNINGS=COMPILATION_NOTES when you compile your Ada subprogram. For example:

```
$ ADA/WARNINGS=COMPILATION_NOTES SWAP

    1   procedure SWAP (A,B: in out INTEGER) is
.......................1
%I, (1) Selected/specified passing mechanism is REFERENCE
.........................2
%I, (2) Selected/specified passing mechanism is REFERENCE
```

Section 5.4 also provides information on the default parameter-passing mechanisms chosen by the VAX Ada compiler; Section 5.8 provides specific information on exporting Ada subprograms that involve parameters passed by descriptor.

NOTE

Some VAX languages allow optional and/or default parameters. Calls from external routines to exported VAX Ada subprograms must supply all parameters that are declared as formal parameters. In particular, an actual value must be supplied even when a default expression is given for a formal parameter in the Ada subprogram specification.

5.3 Conventions for Passing Data in Mixed-Language Programs

When data is passed between routines that are not written in the same VAX language, the calling routine must pass the data in a form and to a location recognized by the routine being called.

In VAX Ada, the manner in which parameters are passed and function results returned is determined by three sets of conventions:

- The semantics of the Ada language
- The linkage conventions required by the VAX Procedure Calling and Condition Handling Standard

- The linkage conventions used by VAX Ada to implement subprogram calls

The following sections discuss these conventions.

5.3.1 Ada Semantics

The Ada language defines two kinds of semantics for parameter passing: copy-in/copy-back semantics and reference semantics.

For parameters of mode **in** or **in out**, *copy-in/copy-back* semantics involves copying the value of the actual parameter into its associated formal parameter at the start of the call; for parameters of mode **in out** or **out**, copy-in/copy-back semantics involves copying the value of the formal parameter back into the actual parameter at the end of the call. *Reference semantics* involves no copies: any modifications to a formal parameter cause the same modifications to happen to the associated actual parameter immediately, and vice versa.

The Ada language requires copy-in/copy-back semantics for scalar and access type parameters (see Chapter 6 of the *VAX Ada Language Reference Manual*). VAX Ada follows these requirements. VAX Ada also uses copy-in/copy-back semantics for address type parameters (parameters of the type SYSTEM.ADDRESS).

The Ada language allows a choice of copy-in/copy-back semantics or reference semantics for array, record, or task type parameters. The VAX Ada compiler takes advantage of this flexibility, and uses either kind of semantics for parameters of these types.

NOTE

When the VAX Ada compiler chooses copy-in/copy-back semantics for a record or array parameter, an update of the formal parameter during the execution of the subprogram does not result in an immediate update of the actual parameter.

VAX Ada implements the Ada semantics for subprogram calls as follows:

1. At the beginning of the subprogram call, if the formal parameter has mode **in** or **in out**, a check is performed to ensure that the actual parameter value satisfies the constraints of the formal parameter. If the actual parameter fails this check, the exception CONSTRAINT_ERROR is raised.
2. If copy-in/copy-back semantics are used, a local variable is allocated to hold the formal parameter.

3. If copy-in/copy-back semantics are used, and if the formal parameter has mode **in** or **in out**, the value of the actual parameter is copied to the formal parameter. In addition, access values and discriminants are copied for mode **out** formal parameters.
4. The subprogram is executed.
5. If copy-in/copy-back semantics are used, and if the formal parameter has mode **in out** or **out**, the value of the formal parameter is copied to the actual parameter.

The exception CONSTRAINT_ERROR may occur at step 4 for record or array parameters when either copy-in/copy-back or reference semantics is used for those parameters. The exception CONSTRAINT_ERROR may also occur at step 5 for any types except record or array types.

5.3.2 VAX Calling Standard Conventions

The primary purpose of the VAX Procedure Calling and Condition Handling Standard is to define how routines are invoked and data is passed between them. Some of the interface attributes that the calling standard specifies follow:

- Stack usage
- Argument list
- Parameter-passing mechanisms
- Function return value
- Register usage

These attributes are examined in more detail in the following sections. The calling standard also defines attributes such as the calling sequence and argument data types and descriptor formats. See *Introduction to VMS System Routines* for more information on these attributes.

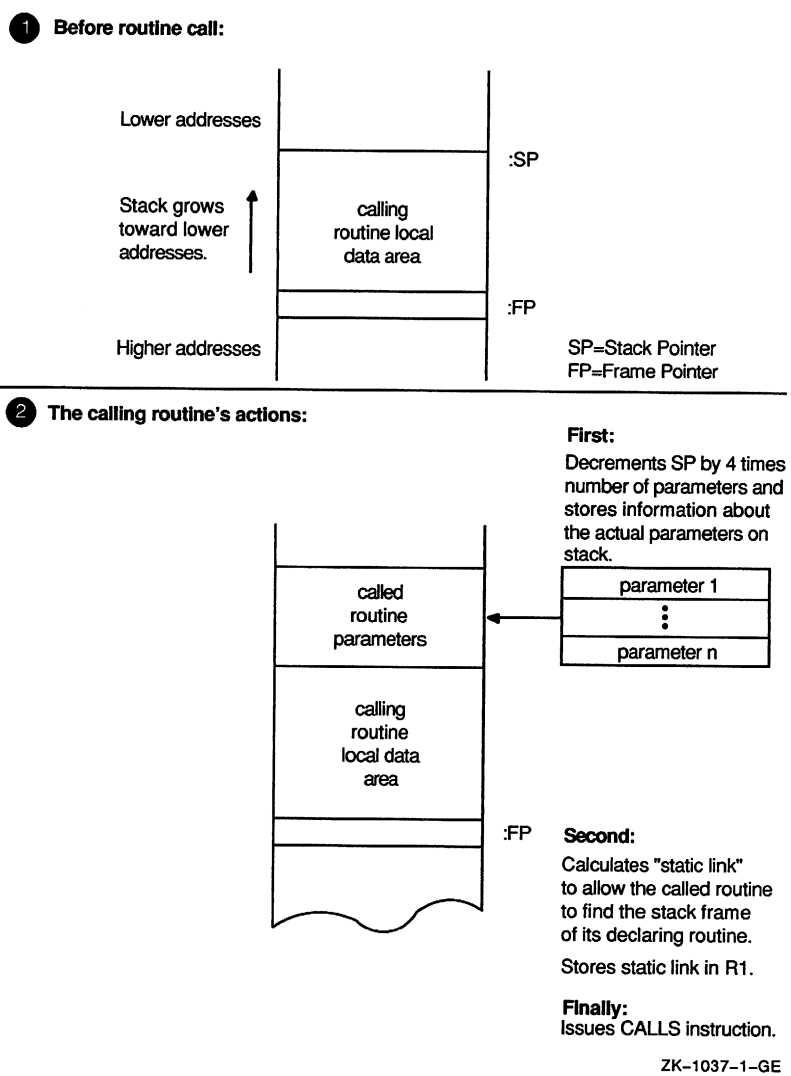
5.3.2.1 The Call Stack

A *call stack* is a last-in-first-out (LIFO) temporary area of storage allocated by the system. In VAX Ada, call stacks are created for the main program and for each task. When an Ada program (or a program written in any language that conforms to the calling standard) is executing, the VAX hardware uses the call stack to maintain information about each routine call. Thus, whenever an Ada program calls a routine (which can be an Ada subprogram or an external routine), or an exported Ada subprogram is called, the hardware creates two structures—an argument list and a call

frame. Depending on the CALL instruction that implements the call, the argument list is placed on the call stack (CALLS), or it is prebuilt and given as an operand (CALLG); the call frame is always placed on the stack. Note that there are special, separate stacks for non-user-mode code; see the *VAX Architecture Reference Manual* for more information.

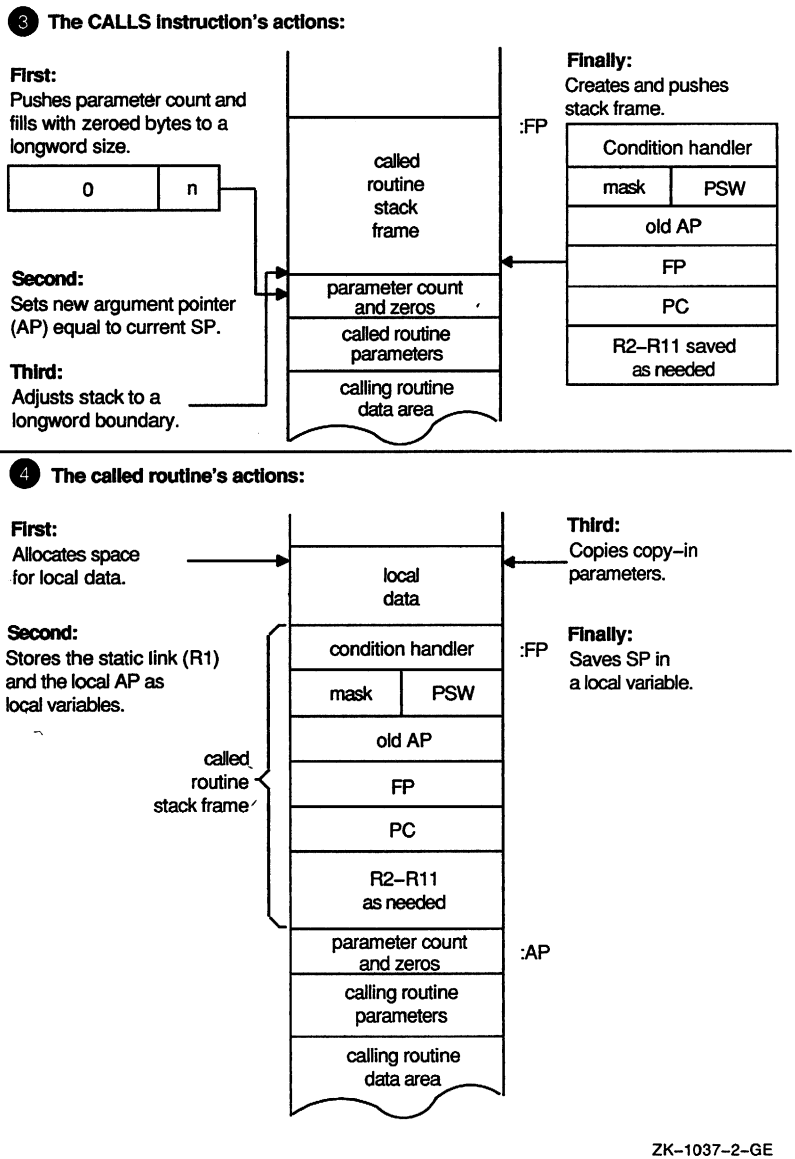
The contents of a call stack during a routine call (CALLS) are shown in Figure 5–1; the argument list and call frame are described in Sections 5.3.2.2 and 5.3.2.5.

Figure 5–1: A Call Stack at Run Time



(continued on next page)

Figure 5–1 (Cont.): A Call Stack at Run Time

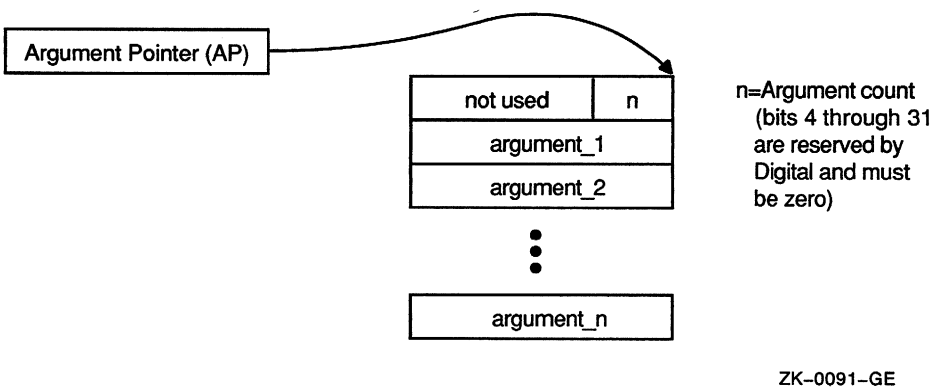


5.3.2.2 The Argument List

The *argument list* is a sequence of longword (32-bit) entries that is pointed to by a register called the argument pointer (AP). The argument list is used to carry information about the actual parameters from the calling routine to the called routine.

The first longword entry contains, in its low byte, an unsigned integer count of the number of arguments in the list; the remaining three bytes must be zero. Each successive longword entry represents one parameter, and, depending on the parameter-passing mechanism being used, can be a 32-bit value, an address of an object, or an address of a descriptor. Figure 5–2 shows the format of an argument list; parameter-passing mechanisms are explained in Section 5.3.2.3.

Figure 5–2: An Argument List



5.3.2.3 Parameter-Passing Mechanisms

The VAX Procedure Calling and Condition Handling Standard defines three mechanisms by which parameters can be passed in the argument list:

- By immediate value—the value of the actual parameter is passed. The argument can be at most 32 bits in length. If the called subprogram or routine expects fewer than 32 bits, it accesses the low-order bits and ignores the high-order bits. When you pass a parameter by value, you pass a copy of the actual parameter value to the routine.

- By reference—The address of a storage location containing the actual parameter value is passed. Note that this storage location may be a copy of the actual parameter.
- By descriptor—The address of a descriptor of the actual parameter is passed; that is, the value passed is the address of a data structure that contains the address of the parameter, along with other information such as the parameter's data type and size.

However, there is no requirement that the compiler use these mechanisms for parameters in subprograms that are not imported or exported. See Section 5.4 for information on the use of these mechanisms for parameters in imported and exported VAX Ada subprograms.

5.3.2.4 Function Return

The VAX Procedure Calling and Condition Handling Standard defines a function as a routine that returns a single value according to the following conventions:

- If the function value requires 32 bits or less, it is returned in register R0.
- If the function value requires from 33 to 64 bits, the low-order bits of the value are returned in register R0, and the high-order bits of the value are returned in register R1.
- If the function value requires more than 64 bits, the calling routine passes an extra parameter—an address—as the first argument in the argument list. The address can point to either the storage for the value or to a descriptor. This kind of function result passing is called the *extra parameter method.*

  If the maximum length of the function value is known, the calling program allocates the required storage and passes either the address of the storage or a descriptor. If the maximum length is not known, the calling program can allocate a descriptor and pass its address. The called function allocates storage for the function value and updates the contents of the descriptor.

See Section 5.5 for information on how VAX Ada interprets these conventions for imported and exported function results.

5.3.2.5 The Call Frame and Register Usage

The VAX Procedure Calling and Condition Handling Standard defines several registers. These registers are defined in Table 5–1.

Table 5–1: VAX Registers

Register	Use
PC	Program counter
SP	Stack pointer
FP	Frame pointer
AP	Argument pointer
R1	Environment value (when necessary)
R0,R1	Function return values

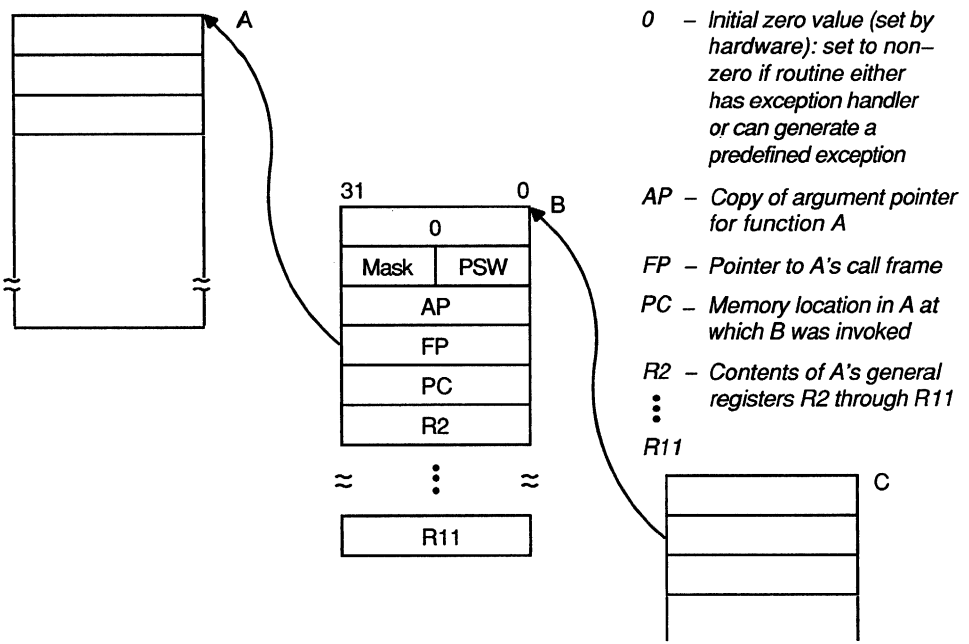
For any called routine, the hardware-created call frame contains the following information:

- A pointer to the call frame of the previous (calling) routine. This pointer is called the saved frame pointer (FP).
- The saved argument pointer (AP) of the previous (calling) routine.
- The address in storage of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the program counter (PC), or saved PC.
- The saved contents of some of the general registers and other control information (such as the condition codes in the processor status word, or PSW, which is the low-order word of the processor status longword, or PSL). Based on a mask specified in the control information, the system restores these registers when control returns to the caller.

The called routine can use registers R0 through R11 and the AP register for computation.

Figure 5–3 shows a call stack with three call frames: routine A calls routine B, and routine B calls routine C. When a routine finishes executing (in Ada, when a function or procedure reaches a return statement or when control reaches the end of a procedure), the system uses the frame pointer in the call frame of the current routine to locate the call frame of the previous (calling) routine. The system then removes the call frame of the current routine from the stack, and returns control to the calling routine (at the point of the saved PC).

Figure 5–3: A Call Stack



5.3.3 VAX Ada Linkage Conventions

Linkage conventions describe the implementation of subprogram calls. The VAX Procedure Calling and Condition Handling Standard defines the general way in which these conventions can be met for all VAX languages (see Section 5.3.2 for a summary of calling standard conventions). VAX Ada interprets these conventions as follows:

- Subprogram calls are made using a VAX CALL (CALLS or CALLG) instruction. The JSB instruction is not used or supported for calls to user-written external routines.
- Parameters are passed in an argument list, except for the first parameters in procedures specified with the pragmas IMPORT_VALUED_PROCEDURE or EXPORT_VALUED_PROCEDURE. The first parameters in these procedures are treated as function results; see Section 5.5.

All other parameters are treated as ordinary parameters, and their mechanisms are determined as described in Sections 5.3.2.3 and 5.4.

- Function results are returned to register R0, registers R0 and R1, or are passed as extra leading parameters in the argument list; see Sections 5.3.2.4 and 5.5. Note that in the case of unconstrained arrays and unconstrained records with discriminants with defaults, the calling routine must pass an area control block; see Sections 5.5.2 and 5.5.3.

5.3.4 The Importance of Data Representation

When writing mixed-language programs, you must make sure that the representation of the data you are passing (as well as the parameter-passing mechanisms) agree. Chapter 2 explains how VAX Ada represents values of the various Ada types. Table 5–3 lists the VAX Ada equivalents for the VAX data types.

For example, when passing strings to non-Ada routines, you must consider any differences that may exist between the Ada string definitions and the other-language string definitions. VAX Ada does not provide a varying string type; instead, you must declare a record like the following:

```
subtype VARYING_STRING_LENGTH is NATURAL range 0..65_535;
type VARYING_STRING (SIZE: VARYING_STRING_LENGTH) is
   record
      STR: STRING(1..SIZE);
   end record;
for VARYING_STRING use
   record
      SIZE at 0 range 0..15;
   end record;
```

Also, the implementation of similar features in another language may differ. For example, the maximum length of a string in VAX Ada is POSITIVE'LAST, or 2^{31}; the maximum length of a string in VAX Pascal is $2^{16} - 1$. Thus, when you pass a string between a VAX Ada subprogram and an imported VAX Pascal routine, you must accommodate for this difference by altering the string type you use in the VAX Ada subprogram. For example:

```
-- Package that declares the string types and printing operation
-- used by the main (Ada) program. The printing operation
-- (VIEW_STRING) is imported from VAX Pascal.
--
package STRING_PACKAGE is
```

```
   -- Declare a short, varying-length string. VARYING_STRING_LENGTH
   -- is needed so that Pascal receives the string correctly.
   --
   subtype VARYING_STRING_LENGTH is NATURAL range 0..65_535;
   type STRING is
      array (VARYING_STRING_LENGTH range <>) of CHARACTER;
   type VARYING_STRING (SIZE: VARYING_STRING_LENGTH) is
      record
         STR: STRING(1..SIZE);
      end record;
   for VARYING_STRING use
      record
         SIZE at 0 range 0..15;
      end record;

   -- Declare a procedure for printing out the string.
   --
   procedure VIEW_STRING(VAR_STR: VARYING_STRING);
   pragma INTERFACE(PASCAL, VIEW_STRING);
   pragma IMPORT_PROCEDURE(INTERNAL        => VIEW_STRING,
                           EXTERNAL        => "View_String",
                           PARAMETER_TYPES => (VARYING_STRING),
                           MECHANISM       => (REFERENCE));

end STRING_PACKAGE;
-------------------------------------------------------------------

-- Main program, which gives a value to and prints out a varying-
-- length string using the Pascal procedure.
--
with STRING_PACKAGE; use STRING_PACKAGE;
procedure MAIN is
   VARYING_STRING_VALUE: VARYING_STRING(8);
begin
   VARYING_STRING_VALUE := (SIZE => 8,
                            STR  => "I'm here");
   VIEW_STRING(VARYING_STRING_VALUE);
end MAIN;
-------------------------------------------------------------------

MODULE View_String (OUTPUT);

TYPE Str = VARYING[65535] OF CHAR;

   [GLOBAL]PROCEDURE View_String (Str_Val: Str);

   BEGIN
      WRITELN(Str_Val);
   END;

END.
```

See Chapter 10 for more information on varying strings.

5.4 VAX Ada Default Parameter-Passing Mechanisms

The calling standard requires that parameters for global routines be passed either by the VAX reference or descriptor mechanisms. Parameters of private or limited private types are passed according to the mechanisms and semantics of their corresponding full type declarations.

The following sections explain the parameter-passing defaults—in terms of both the VAX mechanisms and the Ada semantics—chosen for Ada parameters in imported and exported subprograms. Table 5–3 lists the VAX Ada equivalents for the VAX data types and gives the passing mechanisms that are valid in VAX Ada for those types.

NOTE

The rules for determining VAX Ada passing mechanisms are complex. The /WARNINGS=COMPILATION_NOTES qualifier supplied with any of the compilation commands (DCL ADA or ACS COMPILE or RECOMPILE) gives you the exact mechanism chosen by the compiler for each parameter in an imported or exported Ada subprogram. When writing a specification for an Ada subprogram whose body is imported, you should use an import pragma and the MECHANISM option to explicitly specify (and thus ensure) the passing mechanism for each parameter; see Sections 5.1 and 5.6.

5.4.1 Scalar Type Parameters

Scalar values (enumeration, integer, floating-point, and fixed-point values) are passed by the VAX reference mechanism using Ada copy-in/copy-back semantics.

5.4.2 Array Type Parameters

Array parameter passing is determined by the kind of array being passed:

- A *string* is any one-dimensional array of a discrete type whose components occupy successive, unsigned bytes. (In other words, strings are arrays whose component types include, but are not limited to, the predefined type CHARACTER.)
- A *bit string* is any one-dimensional array of a discrete type whose components occupy successive single bits and are unsigned.

- A *bit array* is any array whose components are not byte aligned, yet which is also not a bit string.
- A *packed array* is any array that has been declared with (and affected by) the pragma PACK (see Section 2.2.1).
- A *packable array* is any bit string or any packed array whose component type is a packable array (not all bit arrays are packable; see Section 2.2.1).

Arrays are usually passed by the VAX descriptor mechanism. The Ada semantics can be either copy-in/copy-back or reference. However, constrained arrays that are neither bit strings nor bit arrays are passed by the VAX reference mechanism (using Ada reference semantics).

Descriptor classes are chosen by the compiler according to how much is known about the array type at compile time. In general, the defaults are those shown in Table 5–2. Additional information about passing arrays by descriptor to imported and exported subprograms is given in Sections 5.6.3 and 5.7.3. See Section 2.1.5 for information on how arrays are represented, and see Section 2.2 for an explanation of how to control that representation.

Table 5–2: Default Descriptor Classes Used by VAX Ada for Array Parameter Passing

Class	Associated Ada Type
DSC$K_CLASS_SB[1]	Any string where the compiler can determine that the number of components will not exceed 65,535.
DSC$K_CLASS_UBSB[1]	Any bit string where the compiler can determine that the number of components will not exceed 65,535.
DSC$K_CLASS_UBA	Any bit array or any other bit string. (CONSTRAINT_ERROR is raised if DSC$W_LENGTH is inadequate.)
DSC$K_CLASS_A	Any other array. (CONSTRAINT_ERROR is raised if DSC$W_LENGTH is inadequate.)

[1]The descriptor classes DSC$K_CLASS_SB and DSC$K_CLASS_UBSB are variants of the VAX DSC$K_CLASS_S and DSC$K_CLASS_UBS descriptors; they have the same fields, plus additional longword fields to hold the lower and upper bounds of the array. All of the VAX descriptors are shown in the *Introduction to VMS System Routines*.

5.4.3 Record Type Parameters

Records are passed by the VAX reference mechanism. The Ada semantics can be either copy-in/copy-back or reference. The address in the argument list refers to a byte-aligned representation. If the actual parameter is a record that is not byte-aligned or that does not occupy an integral number of bytes, the compiler creates a temporary copy that meets those requirements. The address in the argument list then refers to the temporary copy.

If the record type has discriminants with defaults, the calling routine must pass additional information in the argument list in certain cases. In other words, if the formal parameter is unconstrained and has mode **in out** or **out**, the calling routine must provide a "constrainedness bit." The constrainedness bit indicates whether the discriminants of the actual parameter can be changed.

Constrainedness bits are passed by immediate value as an extra longword in the argument list. A subprogram can have up to 32 formal parameters that require a constrainedness bit. The bits are allocated in ascending order, bit 0 being used for the first parameter that requires a constrainedness bit, bit 1 for the second, and so on.

5.4.4 Access Type Parameters

Access values are passed by the VAX reference mechanism using copy-in/copy-back semantics.

The rules for passing access values are the same as those for passing scalar values (see Section 5.4.1), except that any necessary constraint checking is done on the object designated by the access value rather than on the access value itself.

5.4.5 Address Type Parameters

Values of address types (values of type SYSTEM.ADDRESS and its derivatives) are passed by the VAX reference mechanism using Ada copy-in/copy-back semantics.

5.4.6 Task Type Parameters

Values of task types are passed by the VAX reference mechanism. The Ada semantics can be either copy-in/copy-back or reference.

5.4.7 Subprogram Parameters

The Ada language allows you to pass subprograms as parameters only to generic instantiations (see Chapter 12 of the *VAX Ada Language Reference Manual*). You can, however, pass the address of a subprogram by first exporting it (see Section 5.2) and then taking its address with the 'ADDRESS attribute. An exported subprogram must be a library unit or must be declared in the outermost declarative part of a library package. If you try to take the address of a subprogram that is not exported, the compiler issues a warning message and uses the value SYSTEM.ADDRESS_ZERO as the result.

Note that by default VAX Ada passes addresses by reference (see Section 5.4.5).

5.4.8 Entry Parameters

The Ada language allows you to pass task entries as parameters only to generic instantiations (see Chapter 12 of the *VAX Ada Language Reference Manual*).

5.4.9 VAX Ada Equivalents for VAX Data Types

For comparison and reference, Table 5–3 lists the Ada type equivalents and mechanisms chosen for each of the VAX data types defined in the VAX Procedure Calling and Condition Handling Standard.

Table 5–3: VAX Ada Equivalents for VAX Data Types and Their Valid Passing Mechanisms in VAX Ada

Data Type	Symbolic Code	VAX Ada Translation	Passing Mechanism
Absolute date and time	DSC$K_DTYPE_ ADT	STARLET.DATE_ TIME_TYPE	—
Byte integer (signed)	DSC$K_DTYPE_B	SHORT_SHORT_ INTEGER	Reference[1], Descriptor[2]
Bound label value	DSC$K_DTYPE_BLV	Not available	—
Bound procedure value	DSC$K_DTYPE_ BPV	Not available	—
Byte unsigned	DSC$K_DTYPE_BU	Any enumerated type whose values fit into an unsigned byte; SYSTEM.UNSIGNED_ BYTE	Reference[1], Descriptor[2]
COBOL intermediate temporary	DSC$K_DTYPE_CIT	Not available	—
D_floating	DSC$K_DTYPE_D	SYSTEM.D_FLOAT; LONG_FLOAT if pragma LONG_ FLOAT(D_FLOAT) is in effect	Reference[1], Descriptor[2]
D_floating complex	DSC$K_DTYPE_DC	Not available[3]	See Section 5.4.3
Descriptor	DSC$K_DTYPE_ DSC	Not available[3]	See Section 5.4.3
F_floating	DSC$K_DTYPE_F	FLOAT; SYSTEM.F_ FLOAT	Reference[1], Descriptor[2]
F_floating complex	DSC$K_DTYPE_FC	Not available[3]	See Section 5.4.3
G_floating	DSC$K_DTYPE_G	SYSTEM.G_FLOAT; LONG_FLOAT if pragma LONG_ FLOAT(G_FLOAT) is in effect	Reference[1], Descriptor[2]
G_floating complex	DSC$K_DTYPE_GC	Not available[3]	See Section 5.4.3

[1]The default for imported or exported subprograms.
[2]Only when specified as a MECHANISM option of an import pragma.
[3]Can be simulated in VAX Ada with a record type definition.

(continued on next page)

Table 5–3 (Cont.): VAX Ada Equivalents for VAX Data Types and Their Valid Passing Mechanisms in VAX Ada

Data Type	Symbolic Code	VAX Ada Translation	Passing Mechanism
H_floating	DSC$K_DTYPE_H	LONG_LONG_FLOAT; SYSTEM.H_FLOAT	Reference[1], Descriptor[2]
H_floating complex	DSC$K_DTYPE_HC	Not available[3]	See Section 5.4.3
Longword integer (signed)	DSC$K_DTYPE_L	INTEGER	Reference[1], Descriptor[2]
Longword (unsigned)	DSC$K_DTYPE_LU	SYSTEM.UNSIGNED_ LONGWORD	Reference[1], Descriptor[2]
Numeric string, left separate sign	DSC$K_DTYPE_NL	STRING	See Section 5.4.2
Numeric string, left overpunched sign	DSC$K_DTYPE_ NLO	STRING	See Section 5.4.2
Numeric string, right separate sign	DSC$K_DTYPE_NR	STRING	See Section 5.4.2
Numeric string, right overpunched sign	DSC$K_DTYPE_ NRO	STRING	See Section 5.4.2
Numeric string, unsigned	DSC$K_DTYPE_NU	STRING	See Section 5.4.2
Numeric string, zoned sign	DSC$K_DTYPE_NZ	STRING	See Section 5.4.2
Octaword integer (signed)	DSC$K_DTYPE_O	Not available[3]	See Section 5.4.3
Octaword logical (unsigned)	DSC$K_DTYPE_OU	Not available[3]	See Section 5.4.3
Packed decimal string	DSC$K_DTYPE_P	Not available[3]	See Section 5.4.3
Quadword integer (signed)	DSC$K_DTYPE_Q	SYSTEM.UNSIGNED_ QUADWORD, but arithmetic operations are not available	Reference[1], Descriptor[2]

[1]The default for imported or exported subprograms.
[2]Only when specified as a MECHANISM option of an import pragma.
[3]Can be simulated in VAX Ada with a record type definition.

(continued on next page)

Table 5–3 (Cont.): VAX Ada Equivalents for VAX Data Types and Their Valid Passing Mechanisms in VAX Ada

Data Type	Symbolic Code	VAX Ada Translation	Passing Mechanism
Quadword (unsigned)	DSC$K_DTYPE_QU	SYSTEM.UNSIGNED_QUADWORD, but arithmetic operations are not available	Reference[1]
Character string	DSC$K_DTYPE_T	STRING	See Section 5.4.2
Aligned bit string	DSC$K_DTYPE_V	Packed BOOLEAN array	See Section 5.4.2
Varying character string	DSC$K_DTYPE_VT	Not available[3]	See Section 5.4.2
Unaligned bit string	DSC$K_DTYPE_VU	Packed BOOLEAN array	See Section 5.4.2
Word integer (signed)	DSC$K_DTYPE_W	SHORT_INTEGER	Reference[1], Descriptor[2]
Word (unsigned)	DSC$K_DTYPE_WU	Any enumerated type whose values fit into an unsigned word; SYSTEM.UNSIGNED_WORD	Reference[1], Descriptor[2]
Unspecified	DSC$K_DTYPE_Z	Parameter of any type	Depends on Ada type
Procedure entry mask	DSC$K_DTYPE_ZEM	Not available	—
Sequence of instructions	DSC$K_DTYPE_ZI	Not available	—

[1]The default for imported or exported subprograms.
[2]Only when specified as a MECHANISM option of an import pragma.
[3]Can be simulated in VAX Ada with a record type definition.

5.5 VAX Ada Default Function Return Mechanisms

In general, VAX Ada follows the function return conventions summarized in Section 5.3.2.4. The interpretation of those conventions for each of the VAX Ada types is given in the following sections. Note that for any given function result, the size is determined from the result subtype (a subtype is a type

together with a constraint). See Chapter 2 for information on the representation and storage of values of the various Ada types. See Chapter 3 of the *VAX Ada Language Reference Manual* for information on types and subtypes.

5.5.1 Scalar, Access, Address, and Task Type Results

Because they require a maximum of 32 bits, results of enumeration, integer, fixed-point, and F_floating-point types are returned in register R0. Because they require 64 bits, results of floating-point types that use the D_floating or G_floating representation are returned in registers R0 and R1. Results of floating-point types that use the H_floating representation need 128 bits; thus, the calling routine must provide the address of a location that will receive the function result, and the address is passed as the first argument in the argument list.

Results of any access, address, or task type are returned in register R0.

Note that if the allocation of an enumeration or integer type is less than 32 bits, the returned result is zero- or sign-extended (as appropriate) to a clean 32-bit representation.

5.5.2 Array Type Results

The methods by which array function results are returned depend on whether or not the array is constrained.

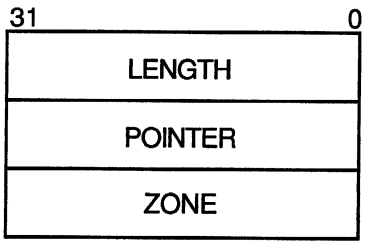
If the type of the function result is a constrained array and all of the following conditions apply, then the function result is returned in register R0 or in registers R0 and R1:

- The array is not a string type (see Section 5.4.2).
- The array bounds and the component size are known at compile time.
- If the array has a size of 32 bits or less, the function result is returned in register R0.
- If the array has a size of 33 to 64 bits, the low-order bits are returned in register R0, and the high-order bits are returned in register R1.

Note that string types are never returned in registers. However, if they are constrained, they can be returned with the extra parameter method (see Section 5.3.2.4).

If the storage size for an array result is not known at compile time or if the size is known to be greater than 64 bits, the calling routine must pass the result using the extra parameter method (see Section 5.3.2.4).

Figure 5–4: Area Control Block Used in Returning Some Function Results



Length (in bytes) of a block of storage for the function return value.

Address of the block of storage.

Zone in which the block of storage is to be freed and/or allocated.

1. If LENGTH is zero, then no storage has been previously allocated, and POINTER is undefined. The called function allocates storage sufficient for the value to be returned using the given ZONE. LENGTH is then set to the size of the block of the storage allocated.

2. If LENGTH is nonzero, then storage has been previously allocated, and POINTER is set to the address of that block of storage. The called function can either reuse the storage (if it is sufficient), or it can deallocate the storage and allocate new storage. (The called routine has the option of doing either if the previously allocated storage is sufficient.)

3. If ZONE is null, then default process dynamic memory is used.

   If zone is –1, then there is no zone associated with the storage, and the calling routine guarantees that the storage described by LENGTH and POINTER is sufficient for the return value.

   Otherwise ZONE is the address of a zone control block.

Note that a single storage area control block can be used in multiple calls without explicit freeing between calls. Also note that by allowing the calling routine to allocate storage when it deems appropriate, the overhead of dynamic memory management is avoided.

ZK–3021–GE

If the function result is of an unconstrained array type (including unconstrained string types), the calling routine must pass the address of an area control block as the first argument in the argument list. The area control block is described and shown in Figure 5–4.

In the case of an unconstrained array type, the area control block is concatenated with an array descriptor. The appropriate descriptor class (DSCK_CLASS_UBSB, DSC_CLASS_UBA, DSC$K_CLASS_SB, or DSC$K_CLASS_A) is chosen according to the rules given in Section 5.4.2 for passing parameters of array types.

The calling routine must initialize the area control block. However, the calling routine does not need to initialize the descriptor or allocate storage for the result. The called function allocates storage in the appropriate zone and fills in the descriptor that refers to the result. The calling routine must release the storage for the result after the storage is used.

5.5.3 Record Type Results

For a simple record type whose size is 32 bits or less, the function result is returned in register R0. If the record size is 33 to 64 bits, the function result is returned in registers R0 and R1. (A *simple record* is one that has no variants and that has components and subcomponents whose constraints are static. Simple records can have discriminants. See Chapter 13 of the *VAX Ada Language Reference Manual* for a complete definition of simple record types and subtypes; see Chapter 4 of the *VAX Ada Language Reference Manual* for a definition of static.)

For a record type whose size is over 64 bits (including large, simple records and constrained records), the calling routine must pass the address of a location to receive the function result as an extra parameter in the argument list (the calling routine must use the extra parameter method; see Section 5.3.2.4). The function returns the result at that location. See Section 2.1.6 for information on how to estimate the storage size of records.

For a function result of an unconstrained record type with discriminants with defaults, the caller must pass an area control block for the called function to use in returning the result. The caller is responsible for releasing the storage for that result after the storage has been used. (The area control block is shown in Figure 5–4.)

5.6 Controlling the Mechanisms for Imported Subprogram Parameters

The MECHANISM option in the VAX Ada import pragmas allows you to control the VAX mechanisms used for passing parameters in imported subprograms. For each parameter, you can specify one of three values—VALUE, REFERENCE, or DESCRIPTOR. For example:

```
package SHOW_MECH is

   type SIMPLE_REC is
      record
         COMP: SHORT_INTEGER;
      end record;

   procedure IMPORT (A: INTEGER; B: STRING; C: SIMPLE_REC);
   pragma INTERFACE (C, IMPORT);
   pragma IMPORT_PROCEDURE (
      INTERNAL        => IMPORT,
      PARAMETER_TYPES => (INTEGER, STRING, SIMPLE_REC),
      MECHANISM       => (REFERENCE, DESCRIPTOR(CLASS => S),
                          VALUE));

end SHOW_MECH;
```

The following sections discuss the use of each of these values in more detail. See Section 5.3.2.3 for a description of the VAX mechanisms that correspond to these values.

NOTE

In addition to the VAX mechanism chosen for a particular parameter (see Section 5.4), the Ada parameter-passing semantics and VAX Ada linkage conventions also apply. The Ada semantics are determined by the parameter's type. See Section 5.3.1 for more information on Ada semantics; see Section 5.3.3 for more information on the VAX Ada linkage conventions.

5.6.1 The VALUE Mechanism Option

The VALUE mechanism option causes the compiler to pass the actual parameter by immediate value. This option has two different effects, depending on the kind of parameter you apply it to:

- When you apply this mechanism to any parameter except the first parameter in a procedure specified with the pragma IMPORT_VALUED_PROCEDURE, the immediate value is passed in the argument list. You

can use this mechanism to pass a parameter of any subtype, as long as it has a compile-time size of 32 bits or less; the associated formal parameter must be of mode **in**.

- When you apply this mechanism to the first parameter in a procedure specified with the pragma IMPORT_VALUED_PROCEDURE, the immediate value is treated as a function result and is returned to registers (see Section 5.7.1). The parameter can have a scalar, access, task, address, simple record, or constrained bit string subtype, but it must have a compile-time size of 64 bits or less. The associated formal parameter must be of mode **out**.

For example:

```
package SHOW_MECHANISMS is

   procedure HYPOTHETICAL(
      RESULT:    out FLOAT;
      BASE:      in INTEGER;
      EXPONENT: in INTEGER);
   pragma INTERFACE (OTHER_LANG, HYPOTHETICAL);
   pragma IMPORT_VALUED_PROCEDURE(
      INTERNAL        => HYPOTHETICAL,
      PARAMETER_TYPES => (FLOAT, INTEGER, INTEGER),
      MECHANISM       => (VALUE, VALUE, VALUE));

end SHOW_MECHANISMS;
```

In this example, all three parameters are passed by the VALUE mechanism. However, the immediate value of RESULT will be returned in register R0; the immediate values of BASE and EXPONENT will be passed in the argument list.

See Chapter 6 for more information on the use of the pragma IMPORT_VALUED_PROCEDURE.

5.6.2 The REFERENCE Mechanism Option

The REFERENCE mechanism causes the address of an actual parameter to be passed in the argument list. You can use this mechanism to pass actual parameters of any type. However, if a non-byte-aligned actual parameter of an array type is passed, the exception CONSTRAINT_ERROR is raised.

You can use the REFERENCE mechanism even in cases where a descriptor or additional information is usually passed (see Sections 5.4.2 and 5.4.3). However, when you use the REFERENCE mechanism in such cases, you must find a way to pass the additional information to the external routine. For example, in VAX Ada you would normally pass an unconstrained array

parameter by descriptor, where the descriptor contains the information about the array bounds. Thus, if you import a routine written in FORTRAN (where all arrays are passed by reference), and pass an unconstrained array parameter, you must pass the array bounds as additional parameters.

Consider the following FORTRAN function:

```
      FUNCTION INNERPROD (A, B, N)

C     This routine multiplies two one-dimensional arrays,
C     element-by-element, then sums the products. Declare A
C     and B as arrays of real numbers.
C
      REAL A(N), B(N)
      SUM = 0
      DO 100 I = 1, N
          SUM = SUM + A(I) * B(I)
100   CONTINUE
      INNERPROD = SUM
      RETURN
      END
```

A and B are adjustable arrays whose bounds are passed as a subprogram argument. To call this routine from Ada, you would declare two unconstrained array parameters to correspond to A and B. By passing the integer parameter N to correspond to the array length parameter, N, in the FORTRAN function, you could then pass the Ada arrays by reference without losing information. The declarations and function call in VAX Ada would be as follows:

```
type ARRAY1 is array (INTEGER range <>) of FLOAT;
function INNERPROD (A, B : ARRAY1; N : INTEGER) return FLOAT;
pragma INTERFACE (FORTRAN, INNERPROD);
pragma IMPORT_FUNCTION (INNERPROD, MECHANISM => REFERENCE);

Q, T : ARRAY1(1..100);
P : FLOAT;
P := INNERPROD (Q, T, Q'LENGTH);
```

Because Q and T are of the same length, either Q'LENGTH or T'LENGTH can be passed to INNERPROD as the actual parameter for N.

5.6.3 The DESCRIPTOR Mechanism Option

The DESCRIPTOR mechanism causes the compiler to pass the address of a string, array, or scalar descriptor, as described in the VAX Procedure Calling and Condition Handling Standard (see the *Introduction to VMS System Routines*). The VAX Ada compiler generates the descriptor and supplies the necessary information.

Parameters of all scalar types, array types, and access types can be passed by descriptor; parameters of record and task types cannot.

The calling standard defines various descriptor classes that can be used for passing parameters of different VAX data types. For certain array parameters, the VAX Ada compiler automatically generates a subset of these descriptors (see Section 5.4.2). For parameters that must be passed by descriptor to an imported routine, the DESCRIPTOR mechanism name can be given with an optional class name argument that specifies the particular descriptor to be used. If you use the DESCRIPTOR mechanism name, but omit the class name, the VAX Ada compiler chooses an appropriate default class depending on the Ada parameter type.

The following example shows the use of the IMPORT_PROCEDURE pragma to call the VMS Run-Time Library routine OTS$CVT_L_TZ, where a string is passed by descriptor. The routine converts a longword (in this case an Ada INTEGER) to a hexadecimal ASCII text string. The routine requires two parameters: an integer passed by reference, and a text string passed by descriptor using descriptor class S.

```
procedure OTSCONVERT is

   INTVALUE : NATURAL;
   HEXSTRING : STRING (1..11);

   procedure CONVERT_TO_HEX (I : in NATURAL; HS : out STRING);
   pragma INTERFACE (RTL, CONVERT_TO_HEX);
   pragma IMPORT_PROCEDURE (INTERNAL  => CONVERT_TO_HEX,
                            EXTERNAL  => "OTS$CVT_L_TZ",
                            MECHANISM => (REFERENCE,
                                          DESCRIPTOR (CLASS => S)));

begin
    . . .
   CONVERT_TO_HEX (INTVALUE, HEXSTRING);
    . . .
end OTSCONVERT;
```

In this example, the formal parameter I is an integer that is passed by reference. The formal parameter HS is an unconstrained string. By default, unconstrained strings are passed by descriptor using descriptor class DSC$K_CLASS_SB. In this case, passing the parameter HS with class DSC$K_CLASS_SB means passing extra information (the string bounds) that the VMS Run-Time Library routine does not need. Thus, for efficiency, the mechanism argument of the IMPORT_PROCEDURE pragma specifies that the second parameter (HS) be passed by descriptor using class DSC$K_CLASS_S. (Because the formal parameter HS is unconstrained, an actual parameter of any length can be passed to the routine.)

Table 5–4 lists the default descriptor classes generated for imported parameters specified with the DESCRIPTOR mechanism. To determine exactly which descriptor class (and descriptor) is generated for imported parameters, use the /WARNINGS=COMPILATION_NOTES qualifier when you compile the Ada specifications for your imported subprograms.

See Section 5.4.2 for definitions of the VAX Ada string, bit-string, and bit-array types.

Table 5–4: Default Descriptor Class Names Used for the DESCRIPTOR Mechanism

Class Name	Use
UBS UBSB	If the parameter is a bit string. (UBS is used only for a constrained bit string whose lower bound is 1.)
A NCA	If the parameter is any array type except a bit string or a bit array.
S SB	If the parameter is of an integer, enumeration, floating-point, fixed-point, access, or address type, or if it is an array of unsigned bytes. (S is used only for a constrained string whose lower bound is 1, or any nonarray type.)
UBA	If the parameter is of any array type.

If you choose to specify class names when you use the DESCRIPTOR mechanism in an import pragma, you must observe the type requirements described in Table 5–5.

Table 5–5: Type Requirements for Descriptor Classes Used by VAX Ada in Importing Routines

Class Name	Requisite Characteristics of Ada Formal Parameter Type
UBS	Unaligned bit string: the base type of the formal parameter must be a one-dimensional array of 1-bit components. If the formal array parameter is constrained, then the lower bound must be equal to 1. A run-time descriptor check occurs to ensure that the actual array parameter has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised.

(continued on next page)

Table 5–5 (Cont.): Type Requirements for Descriptor Classes Used by VAX Ada in Importing Routines

Class Name	Requisite Characteristics of Ada Formal Parameter Type
UBSB	Unaligned bit string with arbitrary bounds: the base type of the formal parameter must be a one-dimensional array of 1-bit components. A run-time descriptor check occurs to ensure that the actual array parameter has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised. The value of A'FIRST'POS need not be equal to 1.
UBA	Unaligned bit array: the base type of the formal parameter must be an array. A run-time descriptor check occurs to ensure that the size of each component of the actual parameter requires no more than 65,535 bits. If this check fails, then CONSTRAINT_ERROR is raised. You normally use this descriptor when the formal parameter array components are unaligned (the formal parameter type has been declared with pragma PACK). If the array components are byte-aligned, use descriptor class A.
S	Scalar or access type, or string: the base type of the formal parameter must be a one-dimensional array of 8-bit unsigned components (a VAX Ada string type; see Section 5.4.2 for a definition of string types), a scalar, or an access type. If the parameter is a constrained array, then the lower bound must be equal to 1. A run-time descriptor check occurs to ensure that the actual array parameter has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised. For a scalar or access type, the DTYPE field of the descriptor is filled in as shown in Table 5–6.
SB	String with arbitrary bounds: the base type of the formal parameter must be a one-dimensional array of unsigned 8-bit components (a VAX Ada string type; see Section 5.4.2 for a definition of string types). A run-time descriptor check occurs to ensure that the actual array parameter has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised.

(continued on next page)

Table 5–5 (Cont.): Type Requirements for Descriptor Classes Used by VAX Ada in Importing Routines

Class Name	Requisite Characteristics of Ada Formal Parameter Type
A	Contiguous array: the base type of the formal parameter must be a byte-aligned array type (that is, an array that starts on a byte boundary) with byte-aligned components or 1-bit components. (This excludes any array of packable components whose component size is not 1, 8, 16, or 32 bits and for which the pragma PACK is given.) If the array type has 1-bit components, a run-time descriptor check is performed to ensure that the actual array parameter is byte-aligned. If this check fails, then CONSTRAINT_ERROR is raised. In all other cases, a run-time descriptor check is performed to ensure that the size of each component does not exceed 65,535 bytes. If this check fails, then CONSTRAINT_ERROR is raised. Note that for a one-dimensional array of unsigned 8-bit components that is not a VAX Ada string type (see Section 5.4.2 for a definition of string types), the descriptor class A can be used instead of class SB because the class A descriptor allows more than 65,535 components to be represented; in other words, class A can be used where it is not known at compile time that there will always be fewer than 65,535 components for all possible values of the type.
NCA	Noncontiguous array: the restrictions on the formal parameter type and the descriptor checks that are performed are the same as for class A. Because VAX Ada never allocates an array of noncontiguous components, this descriptor class is only provided for cases in which the imported routine requires the NCA descriptor.

Table 5–6 lists the data types chosen by the compiler for descriptor DTYPE fields when the DESCRIPTOR mechanism is used in an import pragma. Note that for DSC$K_CLASS_A and DSC$K_CLASS_NCA, the array component type is used to determine the DTYPE, while for all other classes, the formal parameter type is used (that is, the array type is used, except when DSC$K_CLASS_S or DSC$K_CLASS_SB is used to pass nonarrays).

Table 5–6: Descriptor Data Types Used

Type Name	Meaning	Use
DSC$K_DTYPE_VU	Unaligned bit string	Always used for parameters specified with the class names UBS, UBSB, and UBA. Used in particular if the formal parameter type is a bit string or a bit array.
DSC$K_DTYPE_T	Character or character string	Used if the formal parameter type is any enumeration type with a non-negative representation of T'FIRST, with a representation of T'LAST that is less than 256, and with a size of 8 bits. Also used if the formal parameter type is a string of these enumeration type components, which is specified with the class name S or SB.
DSC$K_DTYPE_B DSC$K_DTYPE_BU DSC$K_DTYPE_W DSC$K_DTYPE_WU DSC$K_DTYPE_L DSC$K_DTYPE_LU	Byte integer (signed) Byte logical (unsigned) Word integer (signed) Word logical (unsigned) Longword integer (signed) Longword logical (unsigned)	Used as appropriate for 8-, 16-, or 32-bit components of discrete types.
DSC$K_DTYPE_F DSC$K_DTYPE_D DSC$K_DTYPE_G DSC$K_DTYPE_H	F_floating D_floating G_floating H_floating	Used as appropriate for floating-point types.
DSC$K_DTYPE_Z	Unspecified	Used for all other types.

5.7 Controlling the Return Mechanisms for Imported Function Results

The RESULT_MECHANISM option in the VAX Ada pragma IMPORT_FUNCTION allows you to control the mechanisms used for returning function results in imported functions. You can specify one of three values—VALUE, REFERENCE, or DESCRIPTOR—with this option. For example:

```
package SHOW_RESULT_MECH is
   function CONTROL (X,Y: INTEGER) return STRING;
   pragma INTERFACE (FORTRAN, CONTROL);
   pragma IMPORT_FUNCTION (
      INTERNAL         => CONTROL,
      RESULT_MECHANISM => DESCRIPTOR (CLASS => S));
end SHOW_RESULT_MECH;
```

The following sections discuss the use of these options in more detail.

5.7.1 The VALUE Mechanism Option

The VALUE mechanism option causes the function result to be returned in registers. If the function result subtype has a size of 32 bits or less, it is returned in register R0. If the function result subtype has a size of 33 to 64 bits, the low-order bits are returned in register R0, and the high-order bits are returned in register R1.

This option is valid for scalar, access, task, and address results. It is also valid for simple records and constrained bit strings that have a compile-time size of 64 bits or less.

See Section 5.4.2 for a definition of bit strings; see Section 5.5.3 for a definition of simple records.

5.7.2 The REFERENCE Mechanism Option

The REFERENCE mechanism option causes the address of the function result to be passed by the extra parameter method (the calling—Ada—subprogram adds an extra parameter as the first parameter in the argument list; see Section 5.3.2.4).

This option is valid for any results except unconstrained record or array results.

5.7.3 The DESCRIPTOR Mechanism Option

The DESCRIPTOR mechanism option causes the address of a descriptor for the function result to be passed by the extra parameter method. The choice of descriptors, and the rules for choosing them are the same as those for the parameter-passing DESCRIPTOR mechanism option; see Section 5.6.3.

This option is valid for any results, except task and record results.

In the case of unconstrained arrays, an area control block may be used in addition to a VAX descriptor. See Section 5.5.2 and Section 5.5.3.

5.8 Passing Parameters by Descriptor to Exported Subprograms

When passing parameters by descriptor from an external routine to an exported Ada subprogram, be sure that the calling routine uses the correct descriptor class and fills in the descriptor fields in the manner expected by the Ada subprogram.

To find the correct descriptor class, use the /WARNING=COMPILATION_ NOTES qualifier when you compile the exported Ada subprogram.

When you pass an array using the DSC$K_CLASS_A descriptor for an unconstrained array formal parameter, be sure that the DSC$V_FL_COEFF and DSC$V_FL_BOUNDS bits are set in the DSC$B_AFLAGS field.

When you export an Ada subprogram that would normally receive parameters passed by descriptor class DSC$K_CLASS_SB (see Section 5.4.2), the Ada compiler ensures that parameters passed by descriptor class DSC$K_CLASS_S are also accepted. When a DSC$K_CLASS_S descriptor is received by the exported subprogram, the descriptor bounds are defined as 1..N, where N is the length of the string. The compiler also ensures that DSC$K_CLASS_UBS descriptors are accepted in place of DSC$K_CLASS_ UBSB descriptors, with implicit bounds assumed in the same way. As a result, a slight performance penalty is imposed on exported subprograms where such descriptors are involved.

For example, the following exported Ada function takes a string and a character, and returns the index of the first string component that matches the character:

```
function NFIND (STR: STRING;
                C   : CHARACTER) return INTEGER is
begin

   for I in STR'RANGE loop
      if STR(I) = C then
         return I;
      end if;
   end loop;

   -- If no match, return 0.
   --
   return 0;
```

```
end NFIND;
pragma EXPORT_FUNCTION(NFIND);
```

The following VAX FORTRAN routine uses (imports) the Ada function NFIND:

```
CHARACTER*(12) X
CHARACTER*(1)  B
X = '1234 6789'
B = ' '
N = NFIND(X, %REF(B))
TYPE *, B, N
END
```

In VAX FORTRAN, string parameters are usually passed by descriptor using the DSC$K_CLASS_S descriptor. However, the Ada function expects the string STR parameter to be passed by DSC$K_CLASS_SB descriptor, and the character C to be passed by reference (the CHARACTER type is an enumeration type, which is passed by reference by default in VAX Ada). Because the Ada function is exported, it will also accept the string STR if the string is passed by DSC$K_CLASS_S descriptor. The %REF mechanism specifier in the FORTRAN routine causes B to be passed by reference.

5.9 Sharing Storage with Non-Ada Routines

When you compile a VAX Ada program, the compiler creates up to five contiguous areas of memory, called program sections (psects), to store information about the program. These program sections are named $CODE, $CONSTANT, $DATA, $ADDRESS, and $ZERO. The sections have the following characteristics and properties (see Table 5–7 for a definition of the properties):

$CODE	Contains machine instructions; it has the properties PIC, USR, CON, REL, LCL, SHR, EXE, RD, NOWRT, NOVEC, ALIGN(2).
$CONSTANT	Contains compile-time constants; it has the properties PIC, USR, CON, REL, LCL, SHR, NOEXE, RD, NOWRT, NOVEC, ALIGN(2).
$DATA	Contains static variables (library package data); it has the properties PIC, USR, CON, REL, LCL, NOSHR, NOEXE, RD, WRT, NOVEC, ALIGN(2).

$ADDRESS	Contains address constants, exception vectors, and data produced by the compiler during the course of compilation; it has the properties PIC, USR, CON, REL, LCL, NOSHR, NOEXE, RD, NOWRT, NOVEC, ALIGN(2).
$ZERO	Contains compile-time constants whose values are zero; it has the properties PIC, USR, OVR, REL, LCL, SHR, NOEXE, RD, NOWRT, NOVEC, ALIGN(9).

Except for $CODE, the predefined program sections are not generated unless they are needed.

Table 5–7: Program Section Properties

Class	Meaning
PIC/NOPIC	Position independent or position dependent
USR/LIB	User or library
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute
LCL/GBL	Local or global scope
SHR/NOSHR	Shareable or nonshareable
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable
WRT/NOWRT	Writable or nonwritable
VEC/NOVEC	Vectors or no vectors (protection)
ALIGN	Alignment

When you link your program, the VMS Linker controls memory allocation and sharing according to the properties of each program section. The linker constructs an executable image by dividing the image into sections that have the same properties as their corresponding program sections. If you try to link two program sections that have the same name, but have different properties, the linker issues a warning. To avoid this warning, the VAX languages (including VAX Ada) choose the same properties for equivalent program sections. In other words, the $CODE section generated by the VAX Ada compiler has the same properties as the $CODE section generated by the VAX Pascal compiler, and so on. Thus, the linker allows you to construct a multilanguage image. For more information on the VMS Linker, see the *VMS Linker Utility Manual*; for more information on linking mixed-language programs, see *Developing Ada Programs on VMS Systems*.

The only storage that you can explicitly share with non-Ada programs is storage allocated for objects declared in library packages. VAX Ada provides the pragmas IMPORT_OBJECT, EXPORT_OBJECT, and PSECT_OBJECT to allow such sharing. These pragmas are briefly defined as follows (a detailed definition appears in Chapter 13 of the *VAX Ada Language Reference Manual*). See Chapter 10 for an example of sharing memory between VAX processors.

The syntax of the pragmas IMPORT_OBJECT and EXPORT_OBJECT is as follows:

```
pragma IMPORT_OBJECT | EXPORT_OBJECT
   (internal_name [, external_designator]
      [, [SIZE =>] external_symbol]);

internal_name ::= [INTERNAL =>] simple_name

simple_name ::= identifier

external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal
```

These pragmas are equivalent to the GLOBAL and EXTERNAL attributes in Pascal, the GLOBALDEF and GLOBALREF attributes in PL/I, and the **globaldef** and **globalref** data types in C. You can also use them to share variables with VAX BLISS and MACRO. For example:

```
-- Ada package that declares an imported
-- integer object.
--
package IMPORTOBJ is
   PAS_INT: INTEGER;
   pragma IMPORT_OBJECT(PAS_INT);
end IMPORTOBJ;

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

-- Ada main procedure that prints out the
-- value of the imported object.
--
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with IMPORTOBJ; use IMPORTOBJ;
procedure IMPORTPROC is
begin
  PUT("The value of PAS_INT in Ada is ");
  PUT(PAS_INT);
end IMPORTPROC;

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

```
 { Pascal module that declares and initializes the
   global integer to be imported by the Ada package. }

MODULE Pasobj (INPUT,OUTPUT);
   VAR
      Pas_Int: [GLOBAL] INTEGER := 10;
    . . .
END.
```

This example declares an object in Ada and a variable in Pascal and uses the Ada pragma IMPORT_OBJECT and the Pascal GLOBAL attribute to associate the name Pas_Int with a global symbol. The Pascal module assigns the value 10 to the location referenced by the global symbol, and the Ada main procedure prints it out.

An analogous example of the use of the pragma EXPORT_OBJECT follows:

```
 { Pascal main program that prints out the value
   of the exported Ada object. }

PROGRAM Print_Integer (INPUT,OUTPUT);
   VAR
      Pas_Int: [EXTERNAL] INTEGER;
BEGIN
   WRITE('The value of Pas_Int in PASCAL is ');
   WRITELN(Pas_Int);
END.

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

-- Ada package that declares the object to be
-- exported.
--
package EXPORT_INT is

   PAS_INT: INTEGER := 25;
   pragma EXPORT_OBJECT(PAS_INT);
    . . .

end EXPORT_INT;
```

Here, PAS_INT and its initial value are exported from an Ada package to a Pascal program. PAS_INT is initialized when the Pascal program is executed.

Note that if you want to use the DCL LINK command instead of the ACS LINK command—in other words, you want to link "from" Pascal instead of "from" Ada—you must use the ACS EXPORT command to obtain the Ada object module from the program library. See *Developing Ada Programs on VMS Systems* for more information on linking mixed-language programs.

When sharing variables with VAX BLISS or VAX MACRO, you can use the size option of these pragmas to compare the size of the Ada object with the size expected by the external routine. The internal and external routines both define the same global literal and give as its value the size (in bytes) of the variable that is stored in the program section. (The size option causes the VAX Ada compiler to automatically compute the size.) If the two defined values of the global symbol are not equal, the linker issues an error. Thus, this feature provides some link-time size checking.

For example:

```
! BLISS module that declares a 100-element longword array
! named by the global symbol X and global literal X_SIZE
! that correspond to the imported Ada object and size
! designator of the same names. X_SIZE represents the amount
! of storage allocated for X.
!
module INT_ARRAY (ident = '1-003') =

begin

global
    X : vector[100,long];

global literal
    X_SIZE = %allocation(X);

end
eludom


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

-- Ada package that declares a 105-component array
-- of integers (longwords), makes the object X known
-- to the linker as a global symbol with the pragma
-- IMPORT_OBJECT, and specifies a size designator
-- with the same name as the BLISS global literal
-- in order to obtain link-time consistency checking.
--
package IMPORT_ARRAY is
   X: array (1..105) of INTEGER;
   pragma IMPORT_OBJECT(X, SIZE=>X_SIZE);
end IMPORT_ARRAY;

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

```
-- Ada main procedure that initializes the array
-- object X (all components receive the value 10),
-- and prints out the value of the 12th
-- component.
--
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with IMPORT_ARRAY; use IMPORT_ARRAY;
procedure PRINT_COMPONENT is
begin
   X := (1..105=>10);
   PUT("The value of X(12) is ");
   PUT(X(12));
end PRINT_COMPONENT;
```

The BLISS and Ada code in this example compiles, but when linking is attempted, the linker issues a warning message saying that X_SIZE is multiply defined (it finds two values for X_SIZE). There is also an example of size checking with the pragma PSECT_OBJECT at the end of this section.

The pragma PSECT_OBJECT has the following syntax:

```
pragma PSECT_OBJECT
    (internal_name [, external_designator]
       [, [SIZE =>] external_symbol]);

internal_name ::= [INTERNAL =>] simple_name

simple_name ::= identifier

external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal
```

This pragma is provided primarily for use with FORTRAN or BASIC common blocks, Pascal variables declared with the COMMON or PSECT attribute, EXTERNAL variables in PL/I, or variables declared with the **extern** keyword in C programs. You can also use it to share variables with VAX BLISS and VAX MACRO routines.

Program sections established with the pragma PSECT_OBJECT have the following attributes, which are defined in Table 5–7:

PIC, USR, OVR, REL, GBL, SHR, NOEXE, RD, WRT, NOVEC, ALIGN

Note that the alignment is the greater of the alignment required for the object being exported and ALIGN(2) (that is, longword alignment).

This combination of attributes is the same as the one used by the other languages that allow you to allocate storage in common blocks (that is, overlaid program sections). (Other languages generally have longword alignment.) Note that PSECT_OBJECT program sections are overlaid and global. You cannot change program section attributes in VAX Ada.

Unlike the VAX languages that allow you to store several variables in a particular common block, VAX Ada allows only one object to be allocated in a particular program section. If you want to share storage with FORTRAN common variables, you must declare an Ada record variable in which each component of the record corresponds to one FORTRAN variable. For example:

```
C       FORTRAN declarations:

        INTEGER DAY, MONTH, YEAR
        CHARACTER*20 NAME
        COMMON /BDATE/DAY, MONTH, YEAR / /NAME
        END

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

-- Corresponding VAX Ada declarations:

type DATE is record
     DAY, MONTH, YEAR : INTEGER;
end record;
subtype NAME is STRING(1..20);

BDATE : DATE;
ACCTNAME : NAME;

pragma PSECT_OBJECT (BDATE);
pragma PSECT_OBJECT (ACCTNAME, "$BLANK");
```

This example shows storage allocation in two different program sections. The FORTRAN COMMON statement declares two common blocks. One is named BDATE and contains the three integer variables DAY, MONTH, and YEAR. The second is a blank common block (whose name is $BLANK) and contains the character array variable NAME, which has 20 elements.

The Ada record variable BDATE has three components that correspond to the three variables stored in the common block BDATE. The first pragma PSECT_OBJECT establishes the program section BDATE, which contains the record variable BDATE. (The name of the program section is the same as that of the variable because the psect designator parameter is omitted from the pragma statement.) The Ada variable ACCTNAME is a string of 20 characters, which corresponds to the FORTRAN variable NAME. The second pragma PSECT_OBJECT specifies that storage for ACCTNAME is to be allocated in the program section $BLANK. Note that the psect designator must be a quoted string because the name contains a dollar sign ($).

As with the pragmas IMPORT_OBJECT and EXPORT_OBJECT, you can use the SIZE option with the pragma PSECT_OBJECT to gain some link-time consistency checking when sharing storage with VAX BLISS or MACRO routines. For example:

```
! Declaration of a named psect in VAX BLISS:
PSECT
    NODEFAULT = X(OVERLAY, READ, WRITE, NOEXECUTE);
OWN
    X: PSECT(X) VECTOR[10];

GLOBAL LITERAL
    XLEN = %ALLOCATION(X);

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

-- Corresponding declaration in VAX Ada:

type VECTOR is array(1..10) of INTEGER;
X : VECTOR;

pragma PSECT_OBJECT (X, SIZE => XLEN);
```

The fragment of BLISS code declares a program section named X to store a vector of longwords. The GLOBAL LITERAL statement declares the global symbol XLEN to be equal to the allocation size of the variable X. The Ada code declares an array of integers (which are stored as longwords). The pragma PSECT_OBJECT specifies that the array X is to be stored in the program section named X. Furthermore, the size option directs the Ada compiler to calculate the size of X and declare the global symbol XLEN to be equal to the size of X in bytes. At link time, the linker checks to see that the two declarations of XLEN are equal. If they are not, an error is issued.

Chapter 6

Calling System or Other Callable Routines

VAX Ada provides a variety of features for calling VMS system service, RMS, Run-Time Library, utility, and other callable routines from an Ada program:

- The package STARLET provides VAX Ada types, VAX Ada named numbers representing VMS symbol definitions, and VAX Ada operations for calling VMS system service and RMS routines. The specification of the data types is in Appendix B; the complete specification of this package is in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The package TASKING_SERVICES provides interface routines for calling VMS system services that involve asynchronous system trap (AST) parameters. The specification of this package is in Appendix B, as well as in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The package SYSTEM provides types and operations for manipulating system-related variables and parameters, as well as for obtaining symbol definitions that are not defined in the package STARLET. The specification of this package is described in Chapter 13 of the *VAX Ada Language Reference Manual*, and is given in full in Appendix F of that manual. The specification of this package is also in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The generic package MATH_LIB (and the predefined VAX Ada instantiations FLOAT_MATH_LIB, LONG_FLOAT_MATH_LIB, and LONG_LONG_FLOAT_MATH_LIB) provides routine interfaces for calling many of the VMS Run-Time Library mathematics routines. The specifications for this package and these instantiations are in Appendix B, as well as in the VAX Ada library of predefined units (ADA$PREDEFINED).

- The packages DTK, LIB, MTH, OTS, PPL, SMG, and STR provide VAX Ada types, VAX Ada named numbers representing VMS symbol definitions, and VAX Ada operations for calling VMS Run-Time Library routines. The specifications of these packages are in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The packages CLI, NCS, LBR, and SOR provide VAX Ada types, VAX Ada named numbers representing VMS symbol definitions, and VAX Ada operations for calling VMS Command Language Interpreter, National Character Set, Librarian, and Sort/Merge Utility routines. The specifications of these packages are in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The package CONDITION_HANDLING provides a VAX Ada type for VMS condition values, a set of functions for interpreting condition value components, and a set of interface routines for calling the VMS Run-Time Library routines LIB$MATCH_COND, LIB$STOP, and LIB$SIGNAL. The specification of this package is in Appendix B, as well as in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The package SYSTEM_RUNTIME_TUNING allows you to tune aspects of run-time behavior that are normally controlled by the VAX Ada run-time library. The specification of this package is in Appendix B, as well as in the VAX Ada library of predefined units (ADA$PREDEFINED).
- The VAX Ada import pragmas allow you to write your own interfaces to callable routines; the VAX Ada export pragmas allow you to write Ada subprograms that must be called by or passed as parameters to callable routines (as in the case of call-back routines). These pragmas are discussed in this chapter, in Chapter 5, and in Chapter 13 of the *VAX Ada Language Reference Manual*.

To make copies of the specifications of any of the packages in the library of predefined units (ADA$PREDEFINED), use the ACS EXTRACT SOURCE command. For this command to succeed, either you must have defined ADA$PREDEFINED as your current program library or you must have defined a current Ada program library into which the predefined units have been entered. See *Developing Ada Programs on VMS Systems* for more information. For example:

```
$ ACS EXTRACT SOURCE STARLET
```

The following sections explain how to call VMS system services, Run-Time Library, utility, and other callable routines, and give examples showing the use of the VAX Ada features for accomplishing such calls. You should be familiar with VAX Ada parameter passing and the VAX Procedure Calling and Condition Handling Standard, as well as with the VAX Ada import and

export pragmas. See Chapter 5 of this manual and Chapter 13 of the *VAX Ada Language Reference Manual* for information on these topics.

For specific information on the calling standard and VMS routines, see the appropriate VMS documentation:

- The *Introduction to VMS System Routines* gives general information on VMS system routines, and includes the VAX Procedure Calling and Condition Handling Standard.
- The *VMS System Services Volume* provides information on the VMS system service routines.
- The *VMS Record Management Services Manual* provides information on the VMS RMS routines.
- The *VMS Run-Time Library Routines Volume* provides information on the VMS Run-Time Library routines.
- The *VMS Utility Routines Manual* provides information on the VMS utility routines.

For specific information on callable interfaces for the various VAX layered products, see the products' individual reference manuals.

6.1 Using the VAX Ada System-Routine Packages

The VAX Ada predefined system-routine packages allow you to call system routines directly, without having to specify your own interfaces. The following sections discuss the characteristics and use of these packages.

6.1.1 Parameter Types

As noted in the *Introduction to VMS System Routines*, the VMS environment provides a set of data structures (VMS usages) for denoting the VAX data types used in VMS system, VMS Run-Time Library, and utility routines. Table 6–1 lists these data structures and gives their VAX Ada equivalents. For information on the underlying VAX data types, see Chapter 5; for information on the representation of the VAX Ada data types, see Chapter 2.

NOTE

Many of the equivalents are defined in the packages STARLET and CONDITION_HANDLING. For convenience, the VMS Run-Time Library and utility packages define subtype equivalents for the STARLET and CONDITION_HANDLING types used in those packages.

Table 6–1: VMS Data Structures

VMS Data Structure	VAX Ada Equivalent
access_bit_names	STARLET.ACCESS_BIT_NAMES_TYPE
access_mode	STARLET.ACCESS_MODE_TYPE
address	SYSTEM.ADDRESS
address_range	STARLET.ADDRESS_RANGE_TYPE
arg_list	STARLET.ARG_LIST_TYPE
ast_procedure	SYSTEM.AST_HANDLER
boolean	STANDARD.BOOLEAN
byte_signed	STANDARD.SHORT_SHORT_INTEGER
byte_unsigned	SYSTEM.UNSIGNED_BYTE
channel	STARLET.CHANNEL_TYPE
char_string	STANDARD.STRING
complex_number	User-defined record
cond_value	CONDITION_HANDLING.COND_VALUE_TYPE
context	STARLET.CONTEXT_TYPE
date_time	STARLET.DATE_TIME_TYPE
device_name	STARLET.DEVICE_NAME_TYPE
ef_cluster_name	STARLET.EF_CLUSTER_NAME_TYPE
ef_number	STARLET.EF_NUMBER_TYPE
exit_handler_block	STARLET.EXIT_HANDLER_BLOCK_TYPE
fab	STARLET.FAB_TYPE
file_protection	STARLET.FILE_PROTECTION_TYPE
floating_point	STANDARD.FLOAT STANDARD.LONG_FLOAT STANDARD.LONG_LONG_FLOAT SYSTEM.F_FLOAT SYSTEM.D_FLOAT SYSTEM.G_FLOAT SYSTEM.H_FLOAT
function_code	STARLET.FUNCTION_CODE_TYPE
identifier	SYSTEM.UNSIGNED_LONGWORD
io_status_block	STARLET.IOSB_TYPE

(continued on next page)

Table 6–1 (Cont.): VMS Data Structures

VMS Data Structure	VAX Ada Equivalent
item_list_pair	SYSTEM.UNSIGNED_LONGWORD
item_list_2	STARLET.ITEM_LIST_2_TYPE
item_list_3	STARLET.ITEM_LIST_3_TYPE
item_quota_list	User-defined record
lock_id	STARLET.LOCK_ID_TYPE
lock_status_block	STARLET.LOCK_STATUS_BLOCK_TYPE
lock_value_block	STARLET.LOCK_VALUE_BLOCK_TYPE
logical_name	STARLET.LOGICAL_NAME_TYPE
longword_signed	STANDARD.INTEGER
longword_unsigned	SYSTEM.UNSIGNED_LONGWORD
mask_byte	SYSTEM.UNSIGNED_BYTE
mask_longword	SYSTEM.UNSIGNED_LONGWORD
mask_quadword	SYSTEM.UNSIGNED_QUADWORD
mask_word	SYSTEM.UNSIGNED_WORD
null_arg	SYSTEM.UNSIGNED_LONGWORD
octaword_signed	**array**(1..4) **of** SYSTEM.UNSIGNED_LONGWORD
octaword_unsigned	**array**(1..4) **of** SYSTEM.UNSIGNED_LONGWORD
page_protection	STARLET.PAGE_PROTECTION_TYPE
procedure	SYSTEM.ADDRESS
process_id	STARLET.PROCESS_ID_TYPE
process_name	STARLET.PROCESS_NAME_TYPE
quadword_signed	SYSTEM.UNSIGNED_QUADWORD
quadword_unsigned	SYSTEM.UNSIGNED_QUADWORD
rights_holder	STARLET.RIGHTS_HOLDER_TYPE
rights_id	STARLET.RIGHTS_ID_TYPE
rab	STARLET.RAB_TYPE
section_id	STARLET.SECTION_ID_TYPE
section_name	STARLET.SECTION_NAME_TYPE
system_access_id	STARLET.SYSTEM_ACCESS_ID_TYPE

(continued on next page)

Table 6–1 (Cont.): VMS Data Structures

VMS Data Structure	VAX Ada Equivalent
time_name	STARLET.TIME_NAME_TYPE
uic	STARLET.UIC_TYPE
user_arg	STARLET.USER_ARG_TYPE
varying_arg	SYSTEM.UNSIGNED_LONGWORD
vector_byte_signed	**array**(1..n) **of** STANDARD.SHORT_SHORT_INTEGER
vector_byte_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_BYTE
vector_longword_signed	**array**(1..n) **of** STANDARD.INTEGER
vector_longword_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_LONGWORD
vector_quadword_signed	**array**(1..n) **of** SYSTEM.UNSIGNED_QUADWORD
vector_quadword_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_QUADWORD
vector_word_signed	**array**(1..n) **of** STANDARD.SHORT_INTEGER
vector_word_unsigned	**array**(1..n) **of** SYSTEM.UNSIGNED_WORD
word_signed	**array**(1..n) **of** STANDARD.SHORT_INTEGER
word_unsigned	SYSTEM.UNSIGNED_WORD

6.1.2 Parameter-Passing Mechanisms

The VMS system service, RMS, Run-Time Library, and utility routines conform to the VAX Procedure Calling and Condition Handling Standard. The VAX Ada system-routine packages ensure that the parameters for each routine are passed as required by the routine (by value, by reference, or by descriptor).

See the appropriate VMS documentation for detailed information on the passing mechanisms for parameters of system routines. Table 6–1 lists the VAX Ada equivalents for the VMS data structures. See Chapter 5 for information on passing Ada parameters in mixed-language programs.

NOTE

Any parameter described in the VMS documentation as a routine passed by reference is declared in the VAX Ada packages as a parameter of type ADDRESS. To pass the address of an Ada subprogram, you must first export the subprogram with one of the VAX Ada export pragmas (see Chapter 5 of this manual and Chapter 13 of the *VAX Ada Language Reference Manual*). You

can then use the ADDRESS attribute to obtain the address of the subprogram. An exported subprogram must be a library unit or must be declared at the outermost level of a library package.

6.1.3 Naming Conventions

The following conventions are used in the VAX Ada predefined system-routine packages to form names for named numbers, routine names, and record components:

- In the package STARLET, underscores (_) are used instead of dollar signs ($) because dollar signs are not legal in Ada identifiers. In the VMS Run-Time Library and utility-routine packages, all symbols have had their package-specific prefix removed; for example, you access LIB$SPAWN as LIB.SPAWN.
- Any double underscores are replaced by a single underscore. Leading and trailing underscores are removed.
- If the resulting identifier is an Ada reserved word, the last character is dropped. For example, the system service EXIT becomes EXI, the DTK$TERMINATE routine becomes DTK.TERMINAT, and so on. Other Ada reserved words that are frequently used as record component names are ACCESS and TYPE, which become ACCES and TYP respectively.

See Section 6.1.4 for information on the naming conventions used for record types and initialization constants.

6.1.4 Record Type Declarations

The predefined system-routine packages contain type declarations for VMS control blocks, masks, and so on. For example, the package STARLET declares the following control blocks used by VMS RMS routines:

- The file access block (FAB)
- The record access block (RAB)
- The extended attribute block (XAB)
- The name block (NAM)

Many VMS control blocks have a multilevel structure. For example, the package STARLET represents control blocks by defining a record type for each nested structure. The following record declaration shows a portion of the record type defined in STARLET for the FOP (file-processing options) field of a FAB (VMS RMS file access block); see the *VMS Record*

Management Services Manual for a description of the individual options. The name of the type begins with FAB_ to indicate that FAB_FOP_TYPE is a type declared for a component of a record of type FAB_TYPE.

```
type FAB_FOP_TYPE is
   record
      FILLER_1 : BOOLEAN;
      MXV      : BOOLEAN;
       . . .
      DLT      : BOOLEAN;
       . . .
      FILLER_3 : BOOLEAN;
      ESC      : BOOLEAN;
      TEF      : BOOLEAN;
      OFP      : BOOLEAN;
      KFO      : BOOLEAN;
      FILLER_4 : BOOLEAN;
   end record;
```

FAB_TYPE is declared in STARLET as a record type that contains a component called FOP whose type is FAB_FOP_TYPE:

```
type FAB_TYPE is
   record
      BID: UNSIGNED_BYTE;
      BLN: UNSIGNED_BYTE;
       . . .
      FOP: FAB_FOP_TYPE;
       . . .
   end record;
```

The following example shows how you can access the FOP component:

```
with STARLET;
procedure MODIFY_FOP (FAB1 : in out STARLET.FAB_TYPE;
                      FAB2 : in out STARLET.FAB_TYPE) is
begin

   -- Set the file processing options of FAB1 to
   -- those of FAB2.
   --
   FAB1.FOP := FAB2.FOP;
    . . .
```

```
   -- Set the DLT option to indicate that the file
   -- associated with FAB2 will be deleted when closed.
   --
   FAB2.FOP.DLT := TRUE;

end MODIFY_FOP;
```

An initialization constant is also provided for each record type defined in the predefined system-routine packages to facilitate the initialization of objects of the type. The name of the constant is formed by appending _INIT to the type name. For example, the following declaration is a portion of the STARLET initialization constant for the type FAB_TYPE:

```
FAB_TYPE_INIT : constant FAB_TYPE :=
   (BID => FAB_C_BID,
    BLN => FAB_C_BLN,
     . . .
    FOP => (FILLER_1 => FALSE,
            MXV      => FALSE,
             . . .
            DLT      => FALSE,
             . . . ),
    . . . );
```

A typical use might be as follows:

```
declare

   -- Initialize FAB to contain standard FAB defaults.
   --
   FAB : STARLET.FAB_TYPE := STARLET.FAB_TYPE_INIT;
   STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;

begin
   STARLET.OPEN (STATUS, FAB);
    . . .
end;
```

Likewise, FAB_FOP_TYPE_INIT is defined in STARLET as a constant that you can use to initialize an object or component of the type FAB_FOP_TYPE. A portion of the definition in STARLET is as follows:

```
FAB_FOP_TYPE_INIT : constant FAB_FOP_TYPE :=
   (FILLER_1   => FALSE,
    MXV        => FALSE,
     . . .
    DLT        => FALSE,
     . . .
    FILLER_3   => FALSE,
    ESC        => FALSE,
    TEF        => FALSE,
    OFP        => FALSE,
    KFO        => FALSE,
    FILLER_4   => FALSE);
```

Note that the component names used in this example for the FAB_FOP_TYPE include several that begin with FILLER_. These names in this example and in similar record declarations in the VMS Run-Time Library and utility packages represent reserved fields that are currently unused, but that might be used in the future. The number of reserved fields in any particular record declaration is likely to change from one VMS release to another. Further, the names assigned to the reserved fields are also likely to change. For example, if a component called FILLER_3 were used in a new VMS release, the name of the FILLER_4 component would change to FILLER_3, and FILLER_4 would no longer exist. Thus, you should never explicitly refer to a component that begins with the text FILLER_ in your program. To initialize such components, use the initialization constants declared in the package you are using. For example, to initialize a variable of type FAB_FOP_TYPE, you would write the following:

```
FOP : FAB_FOP_TYPE := FAB_FOP_TYPE_INIT;
```

You can also use FAB_FOP_TYPE_INIT to initialize the FOP component of a FAB. For example:

```
procedure MOD_FOP (FAB : in out STARLET.FAB_TYPE) is
begin
   FAB.FOP := FAB_FOP_TYPE_INIT;
    . . .
end MOD_FOP;
```

Example 6–3 shows the use of some of the VMS RMS control blocks declared in the package STARLET. The example is a program that maps a file to the first available space using the VMS system service SYS$CRMPSC (Create and Map Section) and the VMS RMS routine SYS$OPEN.

6.1.5 Default and Optional Parameters

As discussed in Chapter 5, all native-mode VAX languages, and VMS system service, RMS, Run-Time Library, and utility routines conform to a set of parameter-passing conventions called the VAX Procedure Calling and Condition Handling Standard. In accordance with the standard, each time an Ada subprogram or non-Ada routine is called, an argument list is passed. The first longword of the argument list contains a count of the number of arguments. Each successive longword entry represents one parameter; depending on the parameter-passing mechanism used, that parameter can be a 32-bit value, an address of an object, or an address of a descriptor.

Many VMS system routines provide the notion of an *optional parameter*. By placing a zero in the argument list, you can "omit" an optional parameter that is normally passed by the reference or descriptor mechanism. For example, consider a routine that takes a single optional integer parameter, which is passed by reference. When this routine is called, the first longword contains the value 1, to indicate one argument, and the second longword of the argument list can contain either the value zero, to indicate that the parameter is omitted, or it can contain the address of a memory location containing an integer value.

NOTE

> Passing the value zero by reference (placing in the argument list the address of a memory location that contains the value zero) is different from placing the value zero in the argument list, and is often interpreted differently by the called routine.

Ada provides the notion of a *default parameter expression*. This notion means that you can omit the parameter (specifically only a parameter of mode **in**) in a call, and a default parameter value is automatically supplied. The default parameter expression is evaluated each time the subprogram is called, so it is not feasible for the subprogram body to provide the default value if the value is not present—the default value must be provided as an actual parameter for every call.

The VMS optional-parameter and the Ada default-parameter notions are not equivalent. The VMS system service, RMS, Run-Time Library, and utility routines permit the equivalent of optional **in out** or **out** parameters, but Ada allows only **in** parameters to have default expressions. Further, placing a zero in a VMS argument list to omit an argument can have a different interpretation from a zero passed by reference or a null string passed by descriptor.

Also, VMS system service routines generally require a fixed number of arguments, and you must place a value of zero in the argument list to indicate that an optional parameter has been omitted. VMS RMS, Run-Time Library, and utility routines generally allow optional parameters to be indicated by shortened argument lists.

Thus, the following rules are true for the routines in *all* of the VAX Ada predefined system-routine packages:

- Default or optional **in** parameters that are passed by value are declared with a default, zero value. If you omit a parameter association for one such optional formal parameter, the zero value is placed in the argument list.
- Default or optional **in** parameters that are passed by reference or descriptor to VMS system service routines are declared with a default expression using the VAX Ada NULL_PARAMETER attribute. If you omit a parameter association for one such optional formal parameter, the zero value is placed in the argument list, regardless of the parameter-passing mechanism normally used for the argument.
- Optional **in out** or **out** parameters are overloaded. Two Ada procedure declarations are given for each optional parameter (and the pragma IMPORT_VALUED_PROCEDURE is used to map both Ada subprograms to the same VMS system service). The first declaration specifies the type to be used if an argument is to be passed to the routine; the second specifies the parameter as an **in** parameter of the type ADDRESS to be passed by value, and gives it a default value of ADDRESS_ZERO. If the original parameter is of the type ADDRESS, then the type UNSIGNED_LONGWORD is used for the overloading.

  If the call uses named association, a default argument can be omitted entirely; if it uses positional association, either ADDRESS_ZERO or ADDRESS'NULL_PARAMETER must be specified.

  For routines with multiple **in out** or **out** parameters, overloadings are provided for all combinations, except where two parameters are closely related (for example, a string descriptor is used to hold an output string, and the related parameter is set to the string length).

Because they generally fall at the end of the argument list and can be omitted, optional parameters to the VMS RMS routines in the package STARLET, as well as the VMS Run-Time Library and utility routines, follow one additional rule:

- The FIRST_OPTIONAL_PARAMETER option is used in the pragma IMPORT_VALUED_PROCEDURE to identify the first parameter (of one or of a series of optional parameters) that can be omitted. Then, when

a call to the routine is made, and one or more optional parameters are omitted from the end of the parameter list, a truncated argument list is passed. See the *VAX Ada Language Reference Manual* for more detailed information on the rules for using this mechanism.

In summary, when calling a VMS system service, RMS, Run-Time Library, or utility routine with optional parameters, you should follow these steps:

1. Consult the appropriate VMS system service, RMS, Run-Time Library, or utility routine manual and determine which parameters you want to specify in the call and which you want to omit.
2. Examine the appropriate VAX Ada package for the first routine interface declaration (if it is overloaded) to determine the parameter types. If you are working with the package STARLET, you can also check Appendix B for the declarations of these parameter types.
3. Make the call using named association, giving only the arguments you want to pass.

For example, the SYS$ASSIGN system service routine in the package STARLET has two optional parameters, ACMODE and MBXNAM. The parameter mode for ACMODE is **in** and the passing mechanism is value. Thus, a default value of zero is used to indicate that the value zero is to be placed in the argument list if this parameter is not specified in a call. The parameter mode for MBXNAM is also **in**, but the passing mechanism is descriptor (MBXNAM is of subtype DEVICE_NAME_TYPE, which is a subtype of STRING); thus a default expression of DEVICE_NAME_TYPE'NULL_PARAMETER is used to indicate that the value zero is to be placed in the argument list if this parameter is not specified on a call.

```
--  $ASSIGN
--
--    Assign I/O Channel
--
--      $ASSIGN  devnam ,chan ,[acmode]  ,[mbxnam]
--
--      devnam = address of device name or logical name
--               string descriptor
--      chan   = address of word to receive channel number
--               assigned
--      acmode = access mode associated with channel
--      mbxnam = address of mailbox logical name string
--               descriptor, if mailbox associated with device
--
```

```
procedure ASSIGN (
   STATUS : out COND_VALUE_TYPE;      -- return value
   DEVNAM : in  DEVICE_NAME_TYPE;
   CHAN   : out CHANNEL_TYPE;
   ACMODE : in  ACCESS_MODE_TYPE :=
                   ACCESS_MODE_ZERO; -- 0 value
   MBXNAM : in  DEVICE_NAME_TYPE :=
                   DEVICE_NAME_TYPE'NULL_PARAMETER);
pragma INTERFACE (EXTERNAL, ASSIGN);
pragma IMPORT_VALUED_PROCEDURE (ASSIGN, "SYS$ASSIGN",
   (COND_VALUE_TYPE, DEVICE_NAME_TYPE, CHANNEL_TYPE,
    ACCESS_MODE_TYPE, DEVICE_NAME_TYPE),
   (VALUE, DESCRIPTOR(S), REFERENCE,
    VALUE, DESCRIPTOR(S)));
```

A call to STARLET.ASSIGN that omits the ACMODE parameter, but not the MBXNAM parameter, looks like this (assume that the actual parameters STATUS_VAR, DEVNAM_VAR, CHAN_VAR, and MBXNAM_VAR were previously declared as variables elsewhere in the program):

```
ASSIGN (STATUS => STATUS_VAR,
        DEVNAM => DEVNAM_VAR,
        CHAN   => CHAN_VAR,
        MBXNAM => MBXNAM_VAR);
```

Similarly, the SYS$DEQ system service routine in the package STARLET has four optional parameters: three (LKID, ACMODE, and FLAGS) are **in** parameters passed by value; one (VALBLK) is an **in out** parameter passed by reference. Thus, default values can be provided for LKID, ACMODE, and FLAGS, but an overloading declaration must be provided to allow the VALBLK parameter to be omitted.

```
--  $DEQ
--
--    Dequeue Lock
--
--     $DEQ  [lkid] ,[valblk] ,[acmode] ,[flags]
--
--     lkid   = lock ID of the lock to be dequeued
--
--     valblk = address of the lock value block
--
--     acmode = access mode of the locks to be dequeued
--
--     flags  = optional flags
--
--                   LCK$M_DEQALL
--
```

```
procedure DEQ (
   STATUS : out COND_VALUE_TYPE;       -- return value
   LKID   : in  LOCK_ID_TYPE           := LOCK_ID_ZERO;
   VALBLK : in out LOCK_VALUE_BLOCK_TYPE;
   ACMODE : in  ACCESS_MODE_TYPE       := ACCESS_MODE_ZERO;
   FLAGS  : in  LCK_TYPE               := LCK_TYPE'NULL_PARAMETER);
procedure DEQ (
   STATUS : out COND_VALUE_TYPE;       -- return value
   LKID   : in  LOCK_ID_TYPE           := LOCK_ID_ZERO;
   VALBLK : in  ADDRESS                := ADDRESS_ZERO;
      -- To omit optional VALBLK argument
   ACMODE : in  ACCESS_MODE_TYPE       := ACCESS_MODE_ZERO;
   FLAGS  : in  LCK_TYPE               := LCK_TYPE'NULL_PARAMETER);
pragma INTERFACE (EXTERNAL, DEQ);
pragma IMPORT_VALUED_PROCEDURE (DEQ, "SYS$DEQ",
   (COND_VALUE_TYPE, LOCK_ID_TYPE, LOCK_VALUE_BLOCK_TYPE,
    ACCESS_MODE_TYPE, LCK_TYPE),
   (VALUE, VALUE, REFERENCE, VALUE, VALUE));
pragma IMPORT_VALUED_PROCEDURE (DEQ, "SYS$DEQ",
   (COND_VALUE_TYPE, LOCK_ID_TYPE, ADDRESS,
    ACCESS_MODE_TYPE, LCK_TYPE),
   (VALUE, VALUE, VALUE, VALUE, VALUE));
```

A call to STARLET.DEQ that omits the LKID and ACMODE parameters looks like this (again, assume that the actual parameters were previously defined elsewhere in the program):

```
DEQ (STATUS => STATUS_VAR,
     VALBLK => VALBLK_VAR,
     FLAGS  => FLAGS_VAR);
```

In this case, the first declaration would be used, and default, zero values would be supplied for the omitted LKID and ACMODE parameters.

Alternatively, the following call involves the second declaration, and zeros would automatically be placed in the argument list for the VALBLK, ACMODE, and FLAGS parameters:

```
DEQ (STATUS => STATUS_VAR,
     LKID   => LKID_VAR);
```

The VMS RMS SYS$WRITE routine provides a good example of a STARLET interface for an RMS routine involving optional parameters:

```
--
--  $WRITE
--
--    Write Block to  File
--
--      $WRITE rab, [err], [suc]
--
--          rab = address of rab
--
--          err = address of user error completion routine
--
--          suc = address of user success completion routine
--
procedure WRITE (
   STATUS : out COND_VALUE_TYPE;          -- return value
   RAB    : in out RAB_TYPE;
   ERR    : in  AST_HANDLER  := NO_AST_HANDLER;
   SUC    : in  AST_HANDLER := NO_AST_HANDLER);
pragma INTERFACE (EXTERNAL, WRITE);
pragma IMPORT_VALUED_PROCEDURE (WRITE, "SYS$WRITE",
   (COND_VALUE_TYPE, RAB_TYPE, AST_HANDLER, AST_HANDLER),
   (VALUE, REFERENCE, VALUE, VALUE), ERR);
```

Note that because the two optional parameters (ERR and SUC) are **in** parameters, they have default values; also, the pragma IMPORT_VALUED_ PROCEDURE specifies ERR as the first optional parameter.

The following call involves all four parameters:

```
WRITE (STATUS => STATUS_VAR,
       RAB    => RAB_VAR,
       ERR    => ERR_VAR,
       SUC    => SUC_VAR);
```

The next call omits the two optional parameters, and because the FIRST_ OPTIONAL_PARAMETER mechanism was specified in the routine interface, the argument list will be truncated so that the call involves only the two parameters specified:

```
WRITE (STATUS => STATUS_VAR,
       RAB    => RAB_VAR);
```

If you were to omit ERR, but not SUC, then a zero value would be passed in the argument list for ERR and the argument list would not be truncated.

6.1.6 Calling Asynchronous System Services

Some system services can be executed either synchronously or asynchronously. A *synchronous* service causes your program to wait while the service request is being processed. An *asynchronous* service queues a request and returns control to your program while the request is being processed. When the request is satisfied, the system service uses an AST to interrupt program execution and transfer control to a user-specified procedure. Examples of asynchronous services are SYS$GETJPI and SYS$QIO; their synchronous forms are SYS$GETJPIW and SYS$QIOW. The *VMS System Services Reference Manual* describes these system services in more detail.

You can call asynchronous system services from a VAX Ada program by using tasks and the VAX Ada predefined pragma AST_ENTRY and AST_ENTRY attribute. See Chapter 9 of the *VAX Ada Language Reference Manual* and Chapter 8 for information on tasks and the pragma AST_ENTRY and AST_ENTRY attribute. Chapter 8 also gives several examples of programs where ASTs are handled.

Chapter 8 describes the package TASKING_SERVICES, which provides interface routines for calling services that involve AST parameters (SYSQIO, SYSGETJPI, and so on) from tasks. Note that the subprogam specifications in the package TASKING_SERVICES have Ada bodies, and thus the NULL_PARAMETER attribute could not be used for optional parameters (see Sections 6.1.5 and 6.2.6). As a result, multiple overloadings are used for each combination of optional parameters in the same manner as is done for system services that have optional **in out** or **out** parameters.

The package SYSTEM_RUNTIME_TUNING may also be useful with programs that call asynchronous system services. For example, this package provides operations that allow you to increase the size of the AST packet pool. See Chapter 8 for more information on the AST packet pool and its limitations; see Appendix B for the specification of the package SYSTEM_RUNTIME_TUNING.

6.1.7 Calling Mathematical Routines

VAX Ada provides two packages of operations for calling VMS Run-Time Library mathematical routines:

- The package MATH_LIB—provides interfaces for many of the VMS Run-Time Library mathematical routines and declares exceptions that

can be raised. The interfaces are streamlined for ease of use, rather than exactly matching the VMS Run-Time Library format.

- The package MTH—also provides interfaces for many of the VMS Run-Time Library mathematical routines and declares exceptions that can be raised. The interfaces match the VMS Run-Time Library format (for example, giving separate interfaces for MTH$ACOS, MTH$DCOS, and MTH$GCOS).

The streamlining of the operations in the package MATH_LIB is possible because the package is a generic package that you can instantiate for real types. For convenience, VAX Ada also provides instantiated versions of this package for the types FLOAT, LONG_FLOAT, and LONG_LONG_FLOAT.

For example, you could use the predefined instantiation FLOAT_MATH_LIB as follows:

```
with FLOAT_MATH_LIB;
procedure TRIG_FUNCTIONS is

   X, Y : FLOAT := 3.0;

begin

   -- Test sine-cosine identity.
   --
   Y := FLOAT_MATH_LIB.COS(X)**2 + FLOAT_MATH_LIB.SIN(X)**2;

   -- Find hyperbolic sine two ways.
   --
   Y := FLOAT_MATH_LIB.SINH(X);
   Y := (FLOAT_MATH_LIB.EXP(X) - FLOAT_MATH_LIB.EXP(-X))/2.0;

   -- Find hyperbolic arc sine.
   --
   Y := FLOAT_MATH_LIB.LOG(X + (FLOAT_MATH_LIB.SQRT(X**2 + 1.0)));

end TRIG_FUNCTIONS;
```

If you had declared your own floating-point type, you could declare your own package to instantiate MATH_LIB, and then write a similar procedure as follows:

```
with MATH_LIB;
package MY_FLOATING is

   type MY_FLOATING_TYPE is digits 6;
   package MY_FLOATING_MATH_LIB is
      new MATH_LIB(MY_FLOATING_TYPE);

end MY_FLOATING;
```

```
with MY_FLOATING; use MY_FLOATING;
procedure TRIG_FUNCTIONS is

   X, Y : MY_FLOATING_TYPE := 3.0;

begin

   -- Test sine-cosine identity.
   --
   Y := MY_FLOATING_MATH_LIB.COS(X)**2 +
        MY_FLOATING_MATH_LIB.SIN(X)**2;

   -- Find hyperbolic sine two ways.
   --
   Y := MY_FLOATING_MATH_LIB.SINH(X);
   Y := (MY_FLOATING_MATH_LIB.EXP(X) -
         MY_FLOATING_MATH_LIB.EXP(-X))/2.0;

   -- Find hyperbolic arc sine.
   --
   Y := MY_FLOATING_MATH_LIB.LOG(X +
        (MY_FLOATING_MATH_LIB.SQRT(X**2 + 1.0)));

end TRIG_FUNCTIONS;
```

See Chapter 9 or Appendix A for more information on predefined instantiations.

6.2 Writing Your Own Routine Interfaces

When you need to write your own interface to a callable routine from VAX Ada, you must collect the following information about the routine:

- The name of the routine
- The type of call required
- The data type of each parameter
- The type of access required for each parameter
- The mechanisms needed to pass the parameters
- Whether any of the parameters are themselves routines or the addresses of routines
- Whether or not any parameters are optional

The description of the routine in the appropriate VMS or layered product documentation gives this information.

Then, you must translate this information into Ada terms, write an equivalent Ada subprogram specification, and use the pragma INTERFACE and one of the VAX Ada import pragmas to import the routine so that you can call it as an Ada subprogram.

For example, the system service SYS$TRNLNM (Translate Logical Name) routine has the following format:

```
SYS$TRNLNM [attr],tabnam,lognam[,acmode][,itmlst]
```

The description of this system service indicates the following information:

- The routine returns a condition value and has parameters that may be updated, making this a special type of procedure call in VAX Ada.
- The data types (VMS usages) required are mask_longword (for the *attr* parameter), logical_name (for the *tabnam* and *lognam* parameters), access_mode (for the *acmode* parameter), and item_list_3 (for the *itmlst* parameter). The usage for the condition value returned is cond_value.
- The types of access required are read only (for all parameters) and write only (for the returned condition value).
- The mechanisms needed are reference (for the *attr*, *acmode*, and *itmlst* parameters) and descriptor (for the *tabnam* and *lognam* parameters).
- None of the parameters are themselves routines or addresses of routines.
- The *attr*, *acmode*, and *itmlst* parameters are optional parameters. Note here that SYS$TRNLNM is a VMS system service, and system services require a fixed number of arguments. Thus, the method for omitting each of these parameters from the argument list is to place a zero in the argument list for each omitted parameter, rather than truncating or otherwise altering the list.

The equivalent VAX Ada interface is as follows, assuming that LNM_TYPE, LOGICAL_NAME_TYPE, ACCESS_MODE_TYPE, and ITEM_LIST_3_ TYPE are defined in your program, and that you made use of the predefined packages SYSTEM and CONDITION_HANDLING:

```
procedure TRNLNM (
   STATUS: out CONDITION_HANDLING.COND_VALUE_TYPE;
   ATTR  : in  LNM_TYPE :=
               LNM_TYPE'NULL_PARAMETER;
   TABNAM: in  LOGICAL_NAME_TYPE;
   LOGNAM: in  LOGICAL_NAME_TYPE;
   ACMODE: in  ACCESS_MODE_TYPE :=
               ACCESS_MODE_TYPE'NULL_PARAMETER;
   ITMLST: in  ITEM_LIST_3_TYPE := ITEM_LIST_3_TYPE'NULL_PARAMETER);

pragma INTERFACE (SYSSERV, TRNLNM);
```

```
pragma IMPORT_VALUED_PROCEDURE (
   INTERNAL => TRNLNM,
   EXTERNAL => "SYS$TRNLNM",
   PARAMETER_TYPES =>
      (CONDITION_HANDLING.COND_VALUE_TYPE,
       LNM_TYPE,
       LOGICAL_NAME_TYPE,
       LOGICAL_NAME_TYPE,
       ACCESS_MODE_TYPE,
       ITEM_LIST_3_TYPE),
   MECHANISM =>
      (VALUE,
       REFERENCE,
       DESCRIPTOR(CLASS => S),
       DESCRIPTOR(CLASS => S),
       REFERENCE,
       REFERENCE));
```

The following sections give detailed information on writing VAX Ada interfaces for callable routine interfaces. For more information on the import pragmas and parameter-passing mechanisms, see Chapter 5. For complete examples of interfaces to system routines coded in Ada, see Section 6.5.

6.2.1 Parameter Types

If you are writing your own interface for a VMS routine, see Table 6–1 for a list of the VMS data structures and their VAX Ada equivalents. If you are writing your own interface for another kind of callable routine, see Chapter 5 for information on the VAX Ada equivalents for the VAX data types defined in the VAX Procedure Calling and Condition Handling Standard. For information on the representation of the VAX Ada data types, see Chapter 2.

6.2.2 Determining the Kind of Call

The Ada language provides two kinds of subprograms:

- Procedures, which can have parameters that are updated within the body of the subprogram
- Functions, which return results, but cannot update their parameters

System routines must be imported into an Ada program before they can be called. VAX Ada provides the pragma INTERFACE and the import pragmas IMPORT_PROCEDURE and IMPORT_FUNCTION to allow you to import external routines as procedures and functions, respectively. To pass an Ada procedure or function as a parameter to a system routine, you must first export the system routine (see Section 6.2.5). VAX Ada provides the export pragmas EXPORT_PROCEDURE and EXPORT_FUNCTION to allow you to export Ada subprograms as procedures and functions, respectively.

However, because many system and utility routines return results *and* update their parameters, VAX Ada provides two pragmas designed specifically to import or export subprograms from or to system routines:

- The pragma IMPORT_VALUED_PROCEDURE (in combination with the pragma INTERFACE) allows you to write a VAX Ada interface that will import a routine so that it is interpreted as a procedure in the Ada environment and as a function in the external environment. (For example, all of the routine interfaces in the package STARLET involve the use of this pragma.)
- The pragma EXPORT_VALUED_PROCEDURE allows you to write a VAX Ada interface that will export an Ada procedure so that it is, again, interpreted as a procedure in the Ada environment and as a function in the external environment.

Both pragmas expect the first parameter of the routine or subprogram being imported or exported to receive the result. Thus, the first parameter of the imported or exported "procedure" must be an **out** parameter. The result is returned in this parameter as any function value is returned (see Section 5.3.2.4). You can specify the other parameters with the modes **in**, **in out**, or **out**, according to the actions required by the imported or exported routine or subprogram.

All import and export pragmas involve default parameter-passing mechanisms, as explained in Chapter 5. When you import a system routine, you should explicitly specify the appropriate mechanisms; when you export an Ada subprogram, you must be sure that the calling routine supplies the correct defaults expected by VAX Ada. The /WARNINGS=COMPILATION_ NOTES qualifier for any of the compilation commands (DCL ADA and ACS LOAD, COMPILE, and RECOMPILE) provides diagnostic information about the mechanisms chosen by the compiler for imported and exported subprograms. See *Developing Ada Programs on VMS Systems* for more information on that qualifier and those commands.

When you are working with the pragma EXPORT_VALUED_PROCEDURE, note that the first parameter in a subprogram exported with this pragma is passed by reference if the parameter type is an access type, or a type involving discriminants. This passing mechanism allows parameters of all types to be initialized by the calling routine, as is required by the Ada language for components of an access type or for any discriminants, even in the case of an **out** parameter like the first parameter in a subprogram exported with the pragma EXPORT_VALUED_PROCEDURE.

See Chapter 5 of this manual and Chapter 13 of the *VAX Ada Language Reference Manual* for more information on using the import and export pragmas.

6.2.3 Determining the Access Method

The various kinds of access required by system and utility routine parameters can be translated directly into Ada access modes. Table 6–2 lists the Ada equivalents for the three most common VMS access methods.

Table 6–2: VAX Ada Equivalents for VMS Access Methods

VMS Access Method	VAX Ada Access Mode
Read only	**in**
Write only	**out**
Modify	**in out**

The other access methods—function call (before return), JMP after unwind, call after stack unwind, and call without stack unwind—have no direct VAX Ada equivalents.

Note that when you are using the pragma IMPORT_VALUED_PROCEDURE or the pragma EXPORT_VALUED_PROCEDURE to write a routine interface, the first parameter is reserved for a returned result. Thus, that parameter must have the mode **out**, or an access method of modify (it will usually correspond to a condition value or equivalent returned by the applicable routine or subprogram).

6.2.4 Passing Parameters

Most callable routines (system or layered product) conform to the VAX Procedure Calling and Condition Handling Standard; parameters are passed either by value, by reference, or by descriptor. You should explicitly specify the necessary passing mechanisms in any interface routine you write. See Chapter 5 for more information.

6.2.5 Passing Routines or Subprograms as Parameters

Some system routines take as arguments the addresses of other routines or subprograms (for example, SYS$PUTMSG). To pass an Ada subprogram as a parameter to a system routine, the subprogram must be exported (see the discussion of export pragmas in Chapter 5 and Section 6.2.2). To be exported, a subprogram must be a library unit or must be declared in the outermost declarative part of a library package. You can then pass the subprogram's address to the system routine with the Ada ADDRESS attribute.

If you try to pass the address of a subprogram that is not exported, the compiler issues a warning message.

Example 6–4 has an exported routine that is passed as a parameter to a VMS Run-Time Library routine.

6.2.6 Default and Optional Parameters

To specify a default or optional parameter, choose one of the following methods, depending on the access mode of the parameter:

- For an **in** parameter, use the VAX Ada NULL_PARAMETER attribute, which will place a zero in the argument list, regardless of the passing mechanism used for the argument. For addresses (parameters of type SYSTEM.ADDRESS) that are passed by value and that require default values, use the VAX Ada predefined constant SYSTEM.ADDRESS_ZERO to place a zero value in the argument list. See Chapter 13 of the *VAX Ada Language Reference Manual* for more information about NULL_PARAMETER and ADDRESS_ZERO; see Section 6.1.5 for an explanation of how these mechanisms are used in the VAX Ada predefined system-routine packages.
- For **in out** or **out** parameters, you can use overloading.

- If the routine you are calling allows a truncated argument list, you can also use the FIRST_OPTIONAL_PARAMETER mechanism in whatever import pragma you are using to import the routine. Section 6.1.5 explains how overloading and FIRST_OPTIONAL_PARAMETER are used in the VAX Ada predefined system-routine packages. Chapter 13 of the *VAX Ada Language Reference Manual* gives detailed information on the FIRST_OPTIONAL_PARAMETER mechanism. Note that you can apply the FIRST_OPTIONAL_PARAMETER mechanism only to a formal parameter of mode **in**, and all parameters following that parameter must also be of mode **in**.

6.3 Obtaining Symbol Definitions

Many of the global symbol definitions (condition values, and so on) you will need in calls to system routines are available in the predefined system-routine packages. However, if you need to obtain symbol definitions that are not available from these packages, you can use the following function from the package SYSTEM:

```
function IMPORT_VALUE (SYMBOL : STRING)
   return UNSIGNED_LONGWORD;
```

This function returns the value of the specified (global) symbol. See Chapter 13 of the *VAX Ada Language Reference Manual* for a complete description of its syntax and behavior.

The following example shows the use of the IMPORT_VALUE function to assign the value of the global symbol CMS$_CREATE to the constant CMS_CREATED. A complete example appears in Section 6.5.

```
with SYSTEM;
with CONDITION_HANDLING;
 . . .
procedure CREATE_LIB is

    . . .
   -- Assign a constant the value of the CMS global symbol
   -- CMS$_CREATED, to allow a later check for success or failure.
   --
   CMS_CREATED: constant CONDITION_HANDLING.COND_VALUE_TYPE
      := SYSTEM.IMPORT_VALUE("CMS$_CREATED");
    . . .
```

```
begin
    . . .
   -- Use the imported condition value to check for success.
   --
   if RET_VAL /= CMS_CREATED then
      -- Do something.
   else
      -- Do something else.
   end if;
end CREATE_LIB;
```

6.4 Testing Return Condition Values

Many VMS system service, RMS, Run-Time Library, and utility routines return numeric status values that indicate whether or not they successfully completed the requested operation. The first parameter of all of the routines in the VAX Ada predefined system-routine packages is an **out** parameter, which is set to a status value when the routine finishes execution. This parameter is of the type COND_VALUE_TYPE, which is declared in the predefined package CONDITION_HANDLING.

When a system status value is returned, you can test for success or failure by using one of the condition value evaluation functions provided in the package CONDITION_HANDLING. You can also compare the status value to one of the severity codes declared in the predefined package you are using, or you can compare it to one of the specific condition values that the service returns. You can make the latter comparison by using one of the interface routines for the VMS Run-Time Library routine LIB$MATCH_COND, which are also provided in the package CONDITION_HANDLING.

For example, the following fragment from Example 6–1 uses the CONDITION_HANDLING function SUCCESS to test for successful logical name translation:

```
procedure ORION is
 . . .
RET_STATUS: COND_VALUE_TYPE;
ITEM_LIST : ITEM_LIST_TYPE(1..2);
 . . .
begin
   TRNLNM(STATUS => RET_STATUS,
          TABNAM => "LNM$SYSTEM",
          LOGNAM => "CYGNUS",
          ITMLST => ITEM_LIST);
```

```
   if not CONDITION_HANDLING.SUCCESS(RET_STATUS) then
      -- Raise an error.
   else
      -- Get the name and size and print them out.
   end if;
 . . .
end ORION;
```

Alternatively, you can compare the severity of the status value with one of the following constants (defined in the package STARLET):

STS_K_WARNING
STS_K_SUCCESS
STS_K_ERROR
STS_K_INFO
STS_K_SEVERE

For example:

```
procedure ORION is
 . . .
RET_STATUS: COND_VALUE_TYPE;
ITEM_LIST : ITEM_LIST_TYPE(1..2);
 . . .
begin
   TRNLNM(STATUS => RET_STATUS,
          TABNAM => "LNM$SYSTEM",
          LOGNAM => "CYGNUS",
          ITMLST => ITEM_LIST);
   if CONDITION_HANDLING.SEVERITY(RET_STATUS) /= STS_K_SUCCESS then
      -- Raise an error.
   else
      -- Get the name and size and print them out.
   end if;
 . . .
end ORION;
```

Finally, you can use the function CONDITION_HANDLING.MATCH_COND to test the return status for other condition values (also defined in the package STARLET). For example:

```
with SYSTEM; use SYSTEM;
with STARLET; use STARLET;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with TEXT_IO; use TEXT_IO;
 . . .
```

```
procedure ORION is
    . . .
   RET_STATUS: COND_VALUE_TYPE;
   MATCH_VALUE: INTEGER;
   ITEM_LIST: ITEM_LIST_TYPE(1..2);
   ERROR: exception;
    . . .
begin

   TRNLNM(STATUS => RET_STATUS,
          TABNAM => "LNM$SYSTEM",
          LOGNAM => "CYGNUS",
          ITMLST => ITEM_LIST);
   if not CONDITION_HANDLING.SUCCESS(RET_STATUS)
      then
         -- Locate the error; condition value codes are
         -- given in module $SSDEF in the package STARLET.
         --
         MATCH_VALUE := CONDITION_HANDLING.MATCH_COND (
                           RET_STATUS,
                           SS_IVLOGTAB,
                           SS_NOLOGNAM);

         -- Raise an error exception.
         --
         raise ERROR;
      else
         -- Print out the logical name and its size.
   end if;

   exception
      when ERROR =>
         PUT_LINE("Failed to translate logical name");
         case MATCH_VALUE is
            when 1 => PUT_LINE("TABNAM is not a " &
                               "logical name table");
            when 2 => PUT_LINE("Logical name is not in " &
                               "the name table");
            when others => null;
         end case;

end ORION;
```

To look at the various condition value components, you can use the set of functions provided by the package CONDITION_HANDLING (see Appendix B).

6.5 VMS Routine Examples

Examples 6–1 through 6–7 show the use of the package STARLET and import and export pragmas to make calls to various VMS system service and Run-Time Library routines.

Example 6–1: Calling SYS$TRNLNM Using the Package STARLET

```
with SYSTEM;
with STARLET;
with CONDITION_HANDLING;
with TEXT_IO; use TEXT_IO;
with SHORT_INTEGER_TEXT_IO; use SHORT_INTEGER_TEXT_IO;
procedure ORION is

   -- Declare short string subtype used in retrieving
   -- translated logical name.
   --
   subtype SHORT_STRING is STRING(1..255);

   -- Declare storage for logical name and name size.
   -- Pragma VOLATILE specifies that every read
   -- is to the variables in memory, rather than to
   -- a local copy.
   --
   NAME_BUFFER: SHORT_STRING;
   NAME_SIZE  : SHORT_INTEGER;
   pragma VOLATILE (NAME_BUFFER);
   pragma VOLATILE (NAME_SIZE);

   -- Initialized item list.  Zeros in the last element
   -- indicate the end of the list.
   --
   ITEM_LIST: STARLET.ITEM_LIST_TYPE(1..2) :=
      (1 => (BUF_LEN     => NAME_BUFFER'LENGTH,
             ITEM_CODE   => STARLET.LNM_STRING,
             BUF_ADDRESS => NAME_BUFFER'ADDRESS,
             RET_ADDRESS => NAME_SIZE'ADDRESS),
       2 => (BUF_LEN     => 0,
             ITEM_CODE   => 0,
             BUF_ADDRESS => SYSTEM.ADDRESS_ZERO,
             RET_ADDRESS => SYSTEM.ADDRESS_ZERO));

   -- Variable for receiving returned condition value.
   --
   RET_STATUS: CONDITION_HANDLING.COND_VALUE_TYPE;
```

(continued on next page)

Example 6–1 (Cont.): Calling SYS$TRNLNM Using the Package STARLET

```
begin
   -- Call the system service; default values are
   -- supplied for ATTR and ACMODE.
   --
   STARLET.TRNLNM(STATUS => RET_STATUS,
                  TABNAM => "LNM$SYSTEM",
                  LOGNAM => "CYGNUS",
                  ITMLST => ITEM_LIST);

   -- Logical test for successful or unsuccessful
   -- completion.
   --
   if not CONDITION_HANDLING.SUCCESS(RET_STATUS)
      then
         PUT_LINE("Failed to translate logical name");
      else
         --
         -- Output values
         --
         PUT("Logical name translates to """);
         PUT(NAME_BUFFER(1 .. INTEGER(NAME_SIZE)));
         PUT_LINE("""");
         PUT("Logical name size is ");
         PUT(NAME_SIZE);
         NEW_LINE;
   end if;

end ORION;
```

Example 6–2: Calling SYS$GETQUI Using the Package STARLET

```
-- This program prompts for a queue name (wildcards are acceptable)
-- and displays information on all print jobs in output queues with
-- a job size of 50 blocks or more.  It also displays queue name,
-- job size, user name, and job name information for each job listed.
--

with SYSTEM, STARLET, CONDITION_HANDLING, TEXT_IO, INTEGER_TEXT_IO;
use  SYSTEM, STARLET, CONDITION_HANDLING, TEXT_IO, INTEGER_TEXT_IO;
procedure GETQUI_EXAMPLE is

   QUEUE_ITEM_LIST: ITEM_LIST_TYPE (1..4);
   JOB_ITEM_LIST  : ITEM_LIST_TYPE (1..6);
   ITEM_LIST_END  : ITEM_REC_TYPE := (0,0,ADDRESS_ZERO,ADDRESS_ZERO);
   IOSB           : IOSB_TYPE;

   SEARCH_NAME,
   QUEUE_NAME : STRING (1..31);
   JOB_NAME   : STRING (1..39);
   USER_NAME  : STRING (1..12);

   SEARCH_NAME_LEN: NATURAL;
   QUEUE_NAME_LEN : UNSIGNED_WORD;
   JOB_NAME_LEN,
   USER_NAME_LEN  : UNSIGNED_WORD;

   SEARCH_FLAGS: QUI_SEARCH_FLAGS_TYPE := QUI_SEARCH_FLAGS_TYPE_INIT;
   JOB_STATUS  : QUI_JOB_STATUS_TYPE;
   JOB_SIZE    : INTEGER;

   RET_STATUS_QUEUE, RET_STATUS_JOB : COND_VALUE_TYPE;

begin

   -- Request queue name to search.
   --
   PUT ("Enter queue name to search: ");
   GET_LINE (SEARCH_NAME, SEARCH_NAME_LEN);

   -- Initialize item list for the display queue operation.
   --
   QUEUE_ITEM_LIST := (
      1 => (ITEM_CODE   => QUI_SEARCH_NAME,
            BUF_LEN     => UNSIGNED_WORD(SEARCH_NAME_LEN),
            BUF_ADDRESS => SEARCH_NAME'ADDRESS,
            RET_ADDRESS => ADDRESS_ZERO),
      2 => (ITEM_CODE   => QUI_SEARCH_FLAGS,
            BUF_LEN     => 4,
            BUF_ADDRESS => SEARCH_FLAGS'ADDRESS,
            RET_ADDRESS => ADDRESS_ZERO),
```

(continued on next page)

Example 6–2 (Cont.): Calling SYS$GETQUI Using the Package STARLET

```
    3 => (ITEM_CODE    => QUI_QUEUE_NAME,
          BUF_LEN      => 31,
          BUF_ADDRESS  => QUEUE_NAME'ADDRESS,
          RET_ADDRESS  => QUEUE_NAME_LEN'ADDRESS),
    4 =>  ITEM_LIST_END);

-- Initialize item list for the display job operation.
--
JOB_ITEM_LIST := (
    1 => (ITEM_CODE    => QUI_SEARCH_FLAGS,
          BUF_LEN      => 4,
          BUF_ADDRESS  => SEARCH_FLAGS'ADDRESS,
          RET_ADDRESS  => ADDRESS_ZERO),
    2 => (ITEM_CODE    => QUI_JOB_SIZE,
          BUF_LEN      => 4,
          BUF_ADDRESS  => JOB_SIZE'ADDRESS,
          RET_ADDRESS  => ADDRESS_ZERO),
    3 => (ITEM_CODE    => QUI_JOB_NAME,
          BUF_LEN      => 39,
          BUF_ADDRESS  => JOB_NAME'ADDRESS,
          RET_ADDRESS  => JOB_NAME_LEN'ADDRESS),
    4 => (ITEM_CODE    => QUI_USERNAME,
          BUF_LEN      => 12,
          BUF_ADDRESS  => USER_NAME'ADDRESS,
          RET_ADDRESS  => USER_NAME_LEN'ADDRESS),
    5 => (ITEM_CODE    => QUI_JOB_STATUS,
          BUF_LEN      => 4,
          BUF_ADDRESS  => JOB_STATUS'ADDRESS,
          RET_ADDRESS  => ADDRESS_ZERO),
    6 => ITEM_LIST_END);

-- Request search of all jobs present in output queues; also
-- force wildcard mode to maintain the internal search context
-- block after the first call when a nonwildcard queue name is
-- entered (this action preserves the queue context for the
-- subsequent display job operation).
--
SEARCH_FLAGS.SEARCH_WILDCARD := TRUE;
SEARCH_FLAGS.SEARCH_SYMBIONT := TRUE;
SEARCH_FLAGS.SEARCH_ALL_JOBS := TRUE;

-- Dissolve any internal search context block for the process.
--
GETQUIW (STATUS => RET_STATUS_QUEUE,
FUNC   => QUI_CANCEL_OPERATION);
```

(continued on next page)

Example 6–2 (Cont.): Calling SYS$GETQUI Using the Package STARLET

```
   -- Locate next output queue; loop until an error status is
   -- returned.
   --
   while SUCCESS (RET_STATUS_QUEUE) loop
      GETQUIW (STATUS => RET_STATUS_QUEUE,
               FUNC   => QUI_DISPLAY_QUEUE,
               ITMLST => QUEUE_ITEM_LIST,
               IOSB   => IOSB);
      if SUCCESS (RET_STATUS_QUEUE) then
         RET_STATUS_QUEUE := SEVERITY(IOSB.STATUS);
      end if;
      if SUCCESS (RET_STATUS_QUEUE) then
         NEW_LINE;
         PUT ("Queue name = ");
         PUT_LINE (QUEUE_NAME (1..INTEGER(QUEUE_NAME_LEN)));
         RET_STATUS_JOB := SS_NORMAL;

         -- Get information on next job in queue; loop
         -- until error return.
         --
         while SUCCESS (RET_STATUS_JOB) loop
            GETQUIW (STATUS => RET_STATUS_JOB,
                     FUNC   => QUI_DISPLAY_JOB,
                     ITMLST => JOB_ITEM_LIST,
                     IOSB   => IOSB);
            if SUCCESS (RET_STATUS_JOB) then
               RET_STATUS_JOB := SEVERITY(IOSB.STATUS);
            end if;
            if SUCCESS (RET_STATUS_JOB) and (JOB_SIZE > 50) then
               PUT ("   Job size = ");
               PUT (JOB_SIZE, WIDTH => 5);
               if JOB_STATUS.JOB_INACCESSIBLE then
                  PUT_LINE ("   <no read access privilege>");
               else
                  PUT ("   Username = ");
                  PUT (USER_NAME (1..INTEGER(USER_NAME_LEN)));
                  SET_COL (46);
                  PUT ("   Job name = ");
                  PUT_LINE (JOB_NAME (1..INTEGER(JOB_NAME_LEN)));
               end if;
            end if;
         end loop;
      end if;
   end loop;
end GETQUI_EXAMPLE;
```

Example 6–3: Calling SYS$CRMPSC Using the Package STARLET

```
with SYSTEM; use SYSTEM;
with STARLET;
with CONDITION_HANDLING;
with TEXT_IO; use TEXT_IO;
procedure MAP_FILE is

   NAME : constant STRING := "map_file.ada";
   START_LOC, END_LOC    : ADDRESS;

   FAB      : STARLET.FAB_TYPE := STARLET.FAB_TYPE_INIT;
   XAB      : STARLET.XAB_TYPE(STARLET.XAB_C_FHC)
                  := STARLET.XABFHC_INIT;
   pragma VOLATILE(FAB);
   pragma VOLATILE(XAB);

   STATUS   : CONDITION_HANDLING.COND_VALUE_TYPE;
   CHANNEL : STARLET.CHANNEL_TYPE;

   RETADR,
   INADR    : STARLET.ADDRESS_RANGE_TYPE;

begin

   START_LOC := ADDRESS_ZERO;
   END_LOC := ADDRESS_ZERO;

   -- First, open the file.
   --
   FAB.FNA := NAME'ADDRESS;
   FAB.FNS := NAME'LENGTH;
   FAB.FOP.UFO := TRUE;
   FAB.XAB := XAB'ADDRESS;

   STARLET.OPEN(STATUS, FAB);
```

(continued on next page)

Example 6–3 (Cont.): Calling SYS$CRMPSC Using the Package STARLET

```
   -- Check for the file's existence and, if it exists, that its
   -- format is correct.
   --
   if CONDITION_HANDLING.SEVERITY(STATUS) /= STARLET.STS_K_SUCCESS
   then
      PUT_LINE("Cannot find file");
   else
      if (FAB.ORG /= STARLET.FAB_C_SEQ) or else
         (not FAB.RAT.CR) or else
         (FAB.RFM /= STARLET.FAB_C_VAR)
      then
         PUT_LINE("File is in the wrong format");
      else
         CHANNEL := STARLET.CHANNEL_TYPE(FAB.STV);

         -- Now, map it to the first available space.
         --
         INADR(0) := ADDRESS_ZERO;
         INADR(1) := ADDRESS_ZERO;

          STARLET.CRMPSC(STATUS => STATUS,
                         INADR  => INADR,
                         RETADR => RETADR,
                         FLAGS  => STARLET.SEC_M_EXPREG,
                         CHAN   => CHANNEL);

         -- Check to see if mapping worked; if it did, calculate
         -- the starting and ending points.
         --
         if not CONDITION_HANDLING.SUCCESS(STATUS)
         then
            PUT_LINE("CRMPSC failed");
         else
            START_LOC := RETADR(0);
            if XAB.FFB /= 0
            then
               END_LOC := RETADR(0) + INTEGER(XAB.EBK-1)*512
                          + INTEGER(XAB.FFB);
            else
               END_LOC := RETADR(0) + INTEGER(XAB.EBK)*512;
            end if;
         end if;
      end if;
   end if;
end MAP_FILE;
```

Example 6–4: Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
-- This example uses the following LIB$ routines:
--
-- LIB$FILE_SCAN        Scans a wildcarded file specification,
--                      returning each file.
-- LIB$FILE_SCAN_END    Terminates scan.
--
-- This example contains three compilation units:
--
-- LIB_EXAMPLE_SCAN_SUCCESS     To be called on success of scan.
-- LIB_EXAMPLE_SCAN_FAILURE     To be called on failure of scan.
-- LIB_EXAMPLE                  Main program.
--
-- The subprograms are separate compilation units because they
-- function as callable routines. Because the callback routines
-- are passed as parameters using the ADDRESS attribute, they must
-- be exported. Exported subprograms (routines) must be library
-- subprograms (separate compilation units) or must be declared in
-- a library package.
------------------------------------------------------------------------

-- LIB_EXAMPLE_SCAN_SUCCESS: This procedure is called by every
-- successful lookup of a file from LIB$FILE_SCAN. It is passed the
-- address of the FAB, from whose NAM block the file specification
-- is extracted (starting at the device).
--
with SYSTEM, STARLET, TEXT_IO;
use TEXT_IO;
procedure LIB_EXAMPLE_SCAN_SUCCESS (FAB : STARLET.FAB_TYPE) is
```

(continued on next page)

Example 6–4 (Cont.): Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
    -- Declare the NAM block, and point it to the address in
    -- the FAB.
    --
    NAM : STARLET.NAM_TYPE;
    for NAM use at FAB.NAM;

    -- Declare the length of the string, and determine its starting
    -- position in memory (NAM.L_DEV).
    --
    LEN : constant INTEGER := INTEGER(NAM.B_DEV)+INTEGER(NAM.B_DIR)+
       INTEGER(NAM.B_NAME)+INTEGER(NAM.B_TYPE)+INTEGER(NAM.B_VER);
    STR : STRING (1..LEN);
    for STR use at NAM.L_DEV;

begin
    PUT_LINE (STR);
end LIB_EXAMPLE_SCAN_SUCCESS;

pragma EXPORT_PROCEDURE (
    INTERNAL => LIB_EXAMPLE_SCAN_SUCCESS);

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

-- LIB_EXAMPLE_SCAN_FAILURE: This procedure is called for every
-- failure reported by LIB$FILE_SCAN.
--
with SYSTEM, STARLET, TEXT_IO;
use TEXT_IO;
procedure LIB_EXAMPLE_SCAN_FAILURE (FAB : STARLET.FAB_TYPE) is
begin
    PUT_LINE ("Failure");
end LIB_EXAMPLE_SCAN_FAILURE;

pragma EXPORT_PROCEDURE (
    INTERNAL => LIB_EXAMPLE_SCAN_FAILURE);

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

-- LIB_EXAMPLE: The main program that directs the file scan.
--
with SYSTEM, STARLET, LIB, CONDITION_HANDLING, TEXT_IO;
with LIB_EXAMPLE_SCAN_SUCCESS, LIB_EXAMPLE_SCAN_FAILURE;
use TEXT_IO;

procedure LIB_EXAMPLE is
```

(continued on next page)

Example 6–4 (Cont.): Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
    -- Declare FAB, NAM, buffers, and context.
    --
    MY_FAB : STARLET.FAB_TYPE;
    MY_NAM : STARLET.NAM_TYPE;
    ESS_BUFFER, RSS_BUFFER : STRING (1..STARLET.NAM_C_MAXRSS);
    MY_CONTEXT : LIB.CONTEXT_TYPE;
    STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;

    -- Declare the string to contain the wildcarded list of files
    -- to be searched.
    --
    FILE_SPECIFICATION : constant STRING := "SYS$LIBRARY:*RTL*.*";

    -- Rename "=" to make the code read better.
    --
    function "=" (LEFT, RIGHT : SYSTEM.UNSIGNED_LONGWORD)
       return BOOLEAN
    renames SYSTEM."=";

    -- Import the RMS$_NMF (no more files) value for testing after
    -- the call to LIB$FILE_SCAN.
    --
    RMS_NMF : constant CONDITION_HANDLING.COND_VALUE_TYPE :=
                SYSTEM.IMPORT_VALUE ("RMS$_NMF");

begin

    MY_CONTEXT := 0;

    -- Initialize and set up FAB.
    --
    MY_FAB := STARLET.FAB_TYPE_INIT;
    MY_FAB.FNA := FILE_SPECIFICATION'ADDRESS;
    MY_FAB.FNS := FILE_SPECIFICATION'LENGTH;
    MY_FAB.NAM := MY_NAM'ADDRESS;

    -- Initialize and set up NAM.
    --
    MY_NAM := STARLET.NAM_TYPE_INIT;
    MY_NAM.RSA := RSS_BUFFER'ADDRESS;
    MY_NAM.RSS := SYSTEM.UNSIGNED_BYTE(RSS_BUFFER'LENGTH);
    MY_NAM.ESA := ESS_BUFFER'ADDRESS;
    MY_NAM.ESS := SYSTEM.UNSIGNED_BYTE(ESS_BUFFER'LENGTH);

    -- Output a title.
    --
    PUT_LINE ("Files that match " & FILE_SPECIFICATION & ":");
    NEW_LINE;
```

(continued on next page)

Example 6–4 (Cont.): Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
    -- Scan for the wildcarded files, and handle errors.
    --
    LIB.FILE_SCAN (
       STATUS       => STATUS,
       FAB          => MY_FAB,
       USER_SUCCESS_PROCEDURE => LIB_EXAMPLE_SCAN_SUCCESS'ADDRESS,
       USER_ERROR_PROCEDURE   => LIB_EXAMPLE_SCAN_FAILURE'ADDRESS,
       CONTEXT      => MY_CONTEXT);
    if (STATUS /= RMS_NMF) and then
       (not CONDITION_HANDLING.SUCCESS (STATUS)) then
       CONDITION_HANDLING.SIGNAL (STATUS);
    end if;

    -- Scan done. End it correctly.
    --
    LIB.FILE_SCAN_END (
       STATUS  => STATUS,
       FAB     => MY_FAB,
       CONTEXT => MY_CONTEXT);
    if not CONDITION_HANDLING.SUCCESS (STATUS) then
       CONDITION_HANDLING.SIGNAL (STATUS);
    end if;

end LIB_EXAMPLE;
```

Example 6–5: Calling SMG Routines Using the Package SMG

```
-- This program demonstrates the use of the VAX Ada predefined
-- package SMG. The program uses the SMG.CREATE_MENU and
-- SMG.SELECT_FROM_MENU routines to create an application that
-- uses a vertical menu and allows the user to make multiple
-- selections. When the user exits from the menu, the
-- SMG.DELETE_PASTEBOARD routine clears the user's terminal.
--
with SMG, SYSTEM, CONDITION_HANDLING;
procedure SMG_EXAMPLE is

   subtype STRING_ARRAY_TYPE is STRING(1..9);
   CHOSEN: STRING_ARRAY_TYPE;

   -- To call the SMG.CREATE_MENU routine, you must instantiate
   -- the generic package SMG.CREATE_MENU_PKG. This package
   -- defines both the SMG.CREATE_MENU routine and the type
   -- CHOICES_STRING_ARRAY_TYPE, which is an unconstrained array
   -- of strings.
   --
   package MY_CREATE_MENU is new SMG.CREATE_MENU_PKG(
      LEN => STRING_ARRAY_TYPE'LENGTH);

MENU_CHOICES: MY_CREATE_MENU.CHOICES_STRING_ARRAY_TYPE(1..21) :=
      ("ONE      ", "TWO      ", "THREE    ", "FOUR     ",
       "FIVE     ", "SIX      ", "SEVEN    ", "EIGHT    ",
       "NINE     ", "TEN      ", "ELEVEN   ", "TWELVE   ",
       "THIRTEEN ", "FOURTEEN ", "FIFTEEN  ", "SIXTEEN  ",
       "SEVENTEEN", "EIGHTEEN ", "NINETEEN ", "TWENTY   ",
       "Exit     ");

   RET_STATUS: CONDITION_HANDLING.COND_VALUE_TYPE;
   PASTEBOARD_ID: SYSTEM.UNSIGNED_LONGWORD;
   DISPLAY1_ID, DISPLAY2_ID: SYSTEM.UNSIGNED_LONGWORD;
   KEYBOARD_ID: SYSTEM.UNSIGNED_LONGWORD;
   COUNTER: SYSTEM.UNSIGNED_WORD := 0;

begin

   -- Create the pasteboard on which the virtual displays will
   -- appear.
   --
   SMG.CREATE_PASTEBOARD(
      STATUS                => RET_STATUS,
      PASTEBOARD_ID         => PASTEBOARD_ID);
```

(continued on next page)

Example 6–5 (Cont.): Calling SMG Routines Using the Package SMG

```
-- Create the virtual keyboard to allow input from the user.
--
SMG.CREATE_VIRTUAL_KEYBOARD(
   STATUS                => RET_STATUS,
   KEYBOARD_ID           => KEYBOARD_ID);

-- Create two virtual displays: one for the menu, and one to
-- show the menu choices.
--
SMG.CREATE_VIRTUAL_DISPLAY(
   STATUS                => RET_STATUS,
   NUMBER_OF_ROWS        => 10,
   NUMBER_OF_COLUMNS     => 20,
   DISPLAY_ID            => DISPLAY1_ID,
   DISPLAY_ATTRIBUTES    => SMG.M_BORDER,
   VIDEO_ATTRIBUTES      => SMG.M_BOLD);

SMG.CREATE_VIRTUAL_DISPLAY(
   STATUS                => RET_STATUS,
   NUMBER_OF_ROWS        => 6,
   NUMBER_OF_COLUMNS     => 20,
   DISPLAY_ID            => DISPLAY2_ID,
   DISPLAY_ATTRIBUTES    => SMG.M_BORDER);

-- Paste the virtual displays to the pasteboard (so that they
-- can be seen on the user's terminal).
--
SMG.PASTE_VIRTUAL_DISPLAY(
   STATUS                => RET_STATUS,
   DISPLAY_ID            => DISPLAY2_ID,
   PASTEBOARD_ID         => PASTEBOARD_ID,
   PASTEBOARD_ROW        => 17,
   PASTEBOARD_COLUMN     => 20);

SMG.PASTE_VIRTUAL_DISPLAY(
   STATUS                => RET_STATUS,
   DISPLAY_ID            => DISPLAY1_ID,
   PASTEBOARD_ID         => PASTEBOARD_ID,
   PASTEBOARD_ROW        => 4,
   PASTEBOARD_COLUMN     => 20);
```

(continued on next page)

Example 6–5 (Cont.): Calling SMG Routines Using the Package SMG

```
   -- Create the vertical menu, with its 21 choices ("ONE" through
   -- "TWENTY" and "Exit").
   --
   MY_CREATE_MENU.CREATE_MENU(
      STATUS                => RET_STATUS,
      DISPLAY_ID            => DISPLAY1_ID,
      CHOICES               => MENU_CHOICES,
      MENU_TYPE             => SMG.K_VERTICAL,
      RENDITION_SET         => SMG.M_BOLD,
      RENDITION_COMPLEMENT => SMG.M_BOLD);

   -- Loop while the user chooses items from the menu using the up
   -- and down arrows and the return key; after each choice, the
   -- choice name is output on the screen, and then the default
   -- choice reverts to the first item left on the menu.
   --
   -- The choice "Exit" must be chosen to exit from the menu. When
   -- "Exit" is chosen, the pasteboard and its two displays are
   -- deleted, and program execution is completed.
   --
   while INTEGER(COUNTER) <= 21 loop
      SMG.SELECT_FROM_MENU (
         STATUS                  => RET_STATUS,
         KEYBOARD_ID             => KEYBOARD_ID,
         DISPLAY_ID              => DISPLAY1_ID,
         SELECTED_CHOICE_NUMBER => COUNTER,
         FLAGS                   => SMG.M_REMOVE_ITEM,
         SELECTED_CHOICE_STRING => CHOSEN);

      if CHOSEN = "Exit      " then
         SMG.DELETE_PASTEBOARD (
            STATUS        => RET_STATUS,
            PASTEBOARD_ID => PASTEBOARD_ID);
         exit;
      end if;
      SMG.PUT_LINE(
         STATUS     => RET_STATUS,
         DISPLAY_ID => DISPLAY2_ID,
         TEXT       => CHOSEN);
   end loop;

end SMG_EXAMPLE;
```

Example 6–6: Calling SYS$TRNLNM Using an Import Pragma

```
with SYSTEM;
with CONDITION_HANDLING;
with TEXT_IO; use TEXT_IO;
with SHORT_INTEGER_TEXT_IO; use SHORT_INTEGER_TEXT_IO;
procedure ORION is

   -- Declare short string subtype used in retrieving
   -- translated logical name.
   --
   subtype SHORT_STRING is STRING(1..255);

   -- Declare storage for logical name and name size.
   -- The pragma VOLATILE specifies that every read
   -- of the variables must be to memory, rather
   -- than to a local copy.
   --
   NAME_BUFFER: SHORT_STRING;
   NAME_SIZE  : SHORT_INTEGER;
   pragma VOLATILE(NAME_BUFFER);
   pragma VOLATILE(NAME_SIZE);

   -- Declare subtypes for SYS$TRNLNM parameters.
   --
   subtype LOGICAL_NAME_TYPE is STRING;
   subtype ACCESS_MODE_TYPE is SYSTEM.UNSIGNED_WORD;

   -- Define the VMS item list type.
   --
   type ITEM_REC_TYPE is
      record
         BUF_LEN    : SYSTEM.UNSIGNED_WORD;
         ITEM_CODE  : SYSTEM.UNSIGNED_WORD;
         BUF_ADDRESS: SYSTEM.ADDRESS;
         RET_ADDRESS: SYSTEM.ADDRESS;
      end record;

   type ITEM_LIST_TYPE is
      array (NATURAL range <>) of ITEM_REC_TYPE;

   -- Declare constant representing an item code to
   -- be specified in the item list.
   --
   LNM_STRING : constant := 2;
```

(continued on next page)

Example 6–6 (Cont.): Calling SYS$TRNLNM Using an Import Pragma

```
-- Initialized item list.  Zeros in the last element
-- indicate the end of the list.
--
ITEM_LIST: ITEM_LIST_TYPE(1..2) :=
   (1 => (BUF_LEN     => NAME_BUFFER'LENGTH,
          ITEM_CODE   => LNM_STRING,
          BUF_ADDRESS => NAME_BUFFER'ADDRESS,
          RET_ADDRESS => NAME_SIZE'ADDRESS),
    2 => (BUF_LEN     => 0,
          ITEM_CODE   => 0,
          BUF_ADDRESS => SYSTEM.ADDRESS_ZERO,
          RET_ADDRESS => SYSTEM.ADDRESS_ZERO));

-- Variable for receiving returned condition value.
--
RET_STATUS: CONDITION_HANDLING.COND_VALUE_TYPE;

-- Specify the Ada procedure that corresponds to the
-- system service.
--
procedure TRNLNM (
   STATUS: out CONDITION_HANDLING.COND_VALUE_TYPE;
   ATTR  : in  SYSTEM.UNSIGNED_LONGWORD :=
               SYSTEM.UNSIGNED_LONGWORD'NULL_PARAMETER;
   TABNAM: in  LOGICAL_NAME_TYPE;
   LOGNAM: in  LOGICAL_NAME_TYPE;
   ACMODE: in  ACCESS_MODE_TYPE :=
               ACCESS_MODE_TYPE'NULL_PARAMETER;
   ITMLST: in  ITEM_LIST_TYPE :=
               ITEM_LIST_TYPE'NULL_PARAMETER);
```

(continued on next page)

Example 6–6 (Cont.): Calling SYS$TRNLNM Using an Import Pragma

```
   -- Use the pragmas INTERFACE and IMPORT_VALUED_PROCEDURE to
   -- set up the interface to the actual system service.
   -- Note the specification of parameter-passing mechanisms
   -- by means of the pragma IMPORT_VALUED_PROCEDURE.
   --
   pragma INTERFACE (SYSSERV, TRNLNM);
   pragma IMPORT_VALUED_PROCEDURE (
      INTERNAL => TRNLNM,
      EXTERNAL => "SYS$TRNLNM",
      PARAMETER_TYPES =>
         (CONDITION_HANDLING.COND_VALUE_TYPE,
          SYSTEM.UNSIGNED_LONGWORD,
          LOGICAL_NAME_TYPE,
          LOGICAL_NAME_TYPE,
          ACCESS_MODE_TYPE,
          ITEM_LIST_TYPE),
      MECHANISM =>
         (VALUE,
          REFERENCE,
          DESCRIPTOR(S),
          DESCRIPTOR(S),
          REFERENCE,
          REFERENCE));
begin
   -- Call the system service; default values are
   -- supplied for ATTR and ACMODE.
   --
   TRNLNM(STATUS => RET_STATUS,
          TABNAM => "LNM$SYSTEM",
          LOGNAM => "CYGNUS",
          ITMLST => ITEM_LIST);
```

(continued on next page)

Example 6–6 (Cont.): Calling SYS$TRNLNM Using an Import Pragma

```
   -- Logical test for successful or unsuccessful
   -- completion.
   --
   if not CONDITION_HANDLING.SUCCESS(RET_STATUS)
      then
         PUT_LINE("Failed to translate logical name");
      else
         --
         -- Output values.
         --
         PUT("Logical name translates to """);
         PUT(NAME_BUFFER(1 .. INTEGER(NAME_SIZE)));
         PUT_LINE("""");
         PUT("Logical name size is ");
         PUT(NAME_SIZE);
         NEW_LINE;
   end if;

end ORION;
```

Example 6–7: Using SYSTEM.IMPORT_VALUE to Obtain a Global Symbol Value

```
with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with TEXT_IO; use TEXT_IO;
procedure CREATE_LIB is

   -- Declare the types and objects needed to call
   -- CMS$CREATE_LIBRARY from an Ada program.
   --
   type LIB_DB is array (1..50) of INTEGER;
   subtype DIR_TYPE is STRING (1..14);
   subtype ELEM_TYPE is STRING (1..13);

   LDB: LIB_DB;
   DIR: DIR_TYPE;
   ELEM: ELEM_TYPE;
   RET_VAL: COND_VALUE_TYPE; -- COND_VALUE_TYPE is in the package
                             -- CONDITION_HANDLING.
```

(continued on next page)

Example 6–7 (Cont.): Using SYSTEM.IMPORT_VALUE to Obtain a Global Symbol Value

```
   -- Assign a constant the value of the CMS global symbol
   -- CMS$_CREATED, to allow a later check for success or failure.
   --
   CMS_CREATED: constant COND_VALUE_TYPE :=
      IMPORT_VALUE("CMS$_CREATED");

   -- Declare the interfaces for the callable CMS routines.
   --
   procedure CMS_CREATE_LIBRARY
      (STATUS : out COND_VALUE_TYPE;
       LDB    : in out LIB_DB;
       DIR    : DIR_TYPE);
   pragma INTERFACE (CMS, CMS_CREATE_LIBRARY);
   pragma IMPORT_VALUED_PROCEDURE
      (INTERNAL => CMS_CREATE_LIBRARY,
       EXTERNAL => "CMS$CREATE_LIBRARY",
       PARAMETER_TYPES =>
          (UNSIGNED_LONGWORD,
           LIB_DB,
           DIR_TYPE),
       MECHANISM =>
          (VALUE,
           REFERENCE,
           DESCRIPTOR));

   procedure CMS_CREATE_ELEMENT
      (LDB  : in out LIB_DB;
       ELEM : ELEM_TYPE);
   pragma INTERFACE (CMS, CMS_CREATE_ELEMENT);
   pragma IMPORT_PROCEDURE
      (INTERNAL => CMS_CREATE_ELEMENT,
       EXTERNAL => "CMS$CREATE_ELEMENT",
       PARAMETER_TYPES =>
          (LIB_DB,
           ELEM_TYPE),
       MECHANISM =>
          (REFERENCE,
           DESCRIPTOR));

begin

   -- Initialize the names of the CMS library and element
   -- to be created.
   --
   DIR := "[LENNON.SONGS]";
   ELEM := "LUCY.DIAMONDS";
```

(continued on next page)

Example 6–7 (Cont.): Using SYSTEM.IMPORT_VALUE to Obtain a Global Symbol Value

```
   -- Create the library
   --
   CMS_CREATE_LIBRARY(RET_VAL,LDB,DIR);

   -- Use the imported condition value to check for success.
   --
   if RET_VAL /= CMS_CREATED then
      PUT_LINE("Unsuccessful creation");
   else
      CMS_CREATE_ELEMENT(LDB,ELEM);
   end if;

end CREATE_LIB;
```

Chapter 7

Using the VAX Common Data Dictionary

The VAX Common Data Dictionary (CDD) is a data dictionary system. It allows you to store data definitions so that they can be shared among various VAX languages and VAX data management products. As such, the CDD provides the basis for a highly effective data management system.

The CDD is an optional VAX software product available under a separate license; check with your system manager to determine if it is installed on your system. You should also check to see which version is installed:

- Version 3.4 or lower is called the VAX Common Data Dictionary. It provides a central dictionary, uses the Data Management Utility (DMU) format for internally representing data definitions, and provides the DMU utility, Common Data Dictionary Language (CDDL) compiler, and Dictionary Verify/Fix (CDDV) utility for working with the dictionary and data definitions.
- Version 4.0 or higher is called VAX CDD/Plus. It provides a new set of features, including the ability to create distributed dictionary configurations. It uses a new Common Dictionary Operator (CDO) format for internally representing data definitions, and provides the CDO utility for working with dictionaries and data definitions. However, VAX CDD/Plus is compatible and can be used with DMU dictionaries. VAX CDD/Plus also provides a call interface.

The CDD/Plus documentation explains how to use the CDD. In particular, the *VAX CDD/Plus User's Guide* provides tutorial information on both CDO and DMU dictionaries.

VAX Ada provides a CDD translator utility to allow you to extract CDD data definitions and translate them into Ada source files. By default, a complete Ada package declaration is produced from a CDD data definition; at your option, you can generate a source fragment that you can combine with other fragments using the DCL COPY command or a text editor.

7.1 Using the VAX Ada-from-CDD Translator Utility

When you install VAX Ada, the files you need to use the VAX Ada-from-CDD translator utility are also installed. After VAX Ada is installed, your system will contain the following files:

```
SYS$LIBRARY:ADA$FROM_CDD.CLD
SYS$SYSTEM:ADA$FROM_CDD.EXE
```

In addition, the Ada predefined library (ADA$PREDEFINED) contains the package CDD_TYPES, which you will need to compile the Ada packages or source fragments created by the translator.

Before using the CDD translator, you must define the ADA$FROM_CDD command as follows:

```
$ SET COMMAND SYS$LIBRARY:ADA$FROM_CDD.CLD
```

Once this command is defined, you can call the translator utility as follows:

```
$ ADA$FROM_CDD [/[NO]OUTPUT[=filespec]] [/[NO]PACKAGE] pathname
```

filespec
Is a legal VMS file specification.

pathname
Is a character string that represents the full or relative path name of the CDD data definition to be extracted and translated to Ada. The path name must conform to the rules for forming VAX CDD path names (see the *VAX CDD/Plus User's Guide*). Note that the different dictionary formats use different notation for the dictionary origin:

- For DMU dictionary definitions, a full path name begins with the root name CDD$TOP and specifies the names of all descendants down to the record definition. Descendant names are separated from each other by a period. For example, CDD$TOP.MAIL_ORDER.INFO is a DMU path name for the definition INFO, which is stored in the CDD directory MAIL_ORDER.
- For CDO dictionary definitions, a full path name begins with the dictionary anchor, which specifies the VMS directory where the CDO dictionary hierarchy is stored. The anchor can optionally consist of node, device, and directory components. Descendant names are separated from each other by a period. For example, DISK:[JONES.CDD]MAIL_ORDER.INFO is the CDD path name for the definition INFO, which is stored in the CDD directory MAIL_ORDER.

/OUTPUT (D)
/NOOUTPUT
Specifies the output file; the default is /OUTPUT. If a file specification is not given, a file name is constructed from the CDD path name; SYS$DISK:[ ].ADA is used as the default file specification.

/PACKAGE (D)
/NOPACKAGE
Indicates whether or not the output is to be a complete Ada package declaration; the default is /PACKAGE. When the /PACKAGE qualifier is specified, a complete package is output in the following form:

```
with SYSTEM; use SYSTEM;
with CDD_TYPES; use CDD_TYPES;
package <converted-pathname> is

   <translation of CDD record>

end;
```

When the /NOPACKAGE qualifier is specified, an Ada source fragment containing the translation of the CDD record is output in the following form:

```
<translation of CDD record>
```

7.2 Equivalent VAX Ada and CDDL Data Types

The VAX Ada-from-CDD translator attempts to translate all CDD data types into equivalent VAX Ada data types. However, some CDD data types are not native to VAX Ada. If a data definition contains an unsupported data type, the VAX Ada-from-CDD translator translates it to a bit array or unsigned-byte array (these are defined as subtypes UNSUPPORTED_TYPE1 and UNSUPPORTED_TYPE2 in the package CDD_TYPES), and issues an informational message.

Table 7–1 summarizes the mapping used by the translator between the CDD data types and the equivalent VAX Ada data types. For more information on the CDD data types, see the *VAX CDD/Plus User's Guide*.

The specifications of the packages CDD_TYPES and SYSTEM are given in Appendix B; alternatively, you can obtain the Ada source code for the package CDD_TYPES with the ACS EXTRACT SOURCE command. See *Developing Ada Programs on VMS Systems* for more information on this command.

Table 7–1: Equivalent CDD and VAX Ada Data Types

CDDL Data Type	Ada Data Type
UNSPECIFIED	Unsupported type
SIGNED BYTE	STANDARD.SHORT_SHORT_INTEGER
UNSIGNED BYTE	SYSTEM.UNSIGNED_BYTE
SIGNED WORD	STANDARD.SHORT_INTEGER
UNSIGNED WORD	SYSTEM.UNSIGNED_WORD
SIGNED LONGWORD	STANDARD.INTEGER
UNSIGNED LONGWORD	SYSTEM.UNSIGNED_LONGWORD
SIGNED QUADWORD	SYSTEM.UNSIGNED_QUADWORD
UNSIGNED QUADWORD	SYSTEM.UNSIGNED_QUADWORD
SIGNED OCTAWORD	CDD_TYPES.OCTAWORD_TYPE
UNSIGNED OCTAWORD	CDD_TYPES.OCTAWORD_TYPE
F_FLOATING	STANDARD.FLOAT
F_FLOATING COMPLEX	Unsupported type
D_FLOATING	SYSTEM.D_FLOAT
D_FLOATING COMPLEX	Unsupported type
G_FLOATING	SYSTEM.G_FLOAT
G_FLOATING COMPLEX	Unsupported type
H_FLOATING	STANDARD.LONG_LONG_FLOAT
H_FLOATING COMPLEX	Unsupported type
UNSIGNED NUMERIC	Unsupported type
LEFT OVERPUNCHED NUMERIC	Unsupported type
LEFT SEPARATE NUMERIC	Unsupported type
RIGHT OVERPUNCHED NUMERIC	Unsupported type
RIGHT SEPARATE NUMERIC	Unsupported type
PACKED DECIMAL	Unsupported type
ZONED NUMERIC	Unsupported type

(continued on next page)

Table 7–1 (Cont.): Equivalent CDD and VAX Ada Data Types

CDDL Data Type	Ada Data Type
BIT	One of the subtypes of UNSIGNED_LONGWORD in package SYSTEM (UNSIGNED_1 through UNSIGNED_31); unsupported if larger than 31 bits
DATE	CDD_TYPES.DATE_TIME_TYPE
TEXT	STANDARD.STRING
VARYING STRING	Unsupported type
POINTER	SYSTEM.ADDRESS
VIRTUAL FIELD	Ignored
SEGMENTED STRING	Unsupported type

7.3 Example of Using the Ada-from-CDD Translator

The following example shows the translation of a CDD record definition into an Ada package.

A CDD record definition containing mail order information is extracted and translated from the CDD using the VAX Ada-from-CDD translator. Once the resulting package has been compiled, it can be used by an Ada program that manipulates data based on the type information in the mail order package.

The CDD record definition is as follows:

NOTE

For the purpose of illustration, this definition is written in the CDD Data Definition Language (CDDL) used with DMU dictionaries; you can construct a similar record definition using most of the same statements with the VAX CDD/Plus CDO utility. See the *VAX CDD/Plus User's Guide* for information on the CDO utility.

```
DEFINE RECORD CDD$TOP.MAIL_ORDER.INFO.
    MAIL_ORDER STRUCTURE.
       ORDER_NUM       DATATYPE IS LONGWORD.
       NAME            DATATYPE IS TEXT
                       SIZE IS 20 CHARACTERS.
       ADDRESS         DATATYPE IS TEXT
                       SIZE IS 20 CHARACTERS.
       CITY            DATATYPE IS TEXT
                       SIZE IS 19 CHARACTERS.
       STATE           DATATYPE IS TEXT
                       SIZE IS 2 CHARACTERS.
       ZIP_CODE        DATATYPE IS TEXT
                       SIZE IS 5 CHARACTERS.
       ITEM_NUM        DATATYPE IS LONGWORD.
       SHIPPING        DATATYPE IS F_FLOATING.
    END MAIL_ORDER STRUCTURE.
END MAIL_ORDER.INFO RECORD.
```

To translate this definition to an Ada package, you first define the ADA$FROM_CDD command, and then execute the command so that it extracts and translates the CDD record (assumed in this example to have a DMU path name of CDD$TOP.MAIL_ORDER.INFO). For example:

```
$ SET COMMAND SYS$LIBRARY:ADA$FROM_CDD.CLD
$ ADA$FROM_CDD/OUTPUT=INFO.ADA/PACKAGE CDD$TOP.MAIL_ORDER.INFO
```

You need to define the ADA$FROM_CDD command only once for any given terminal session; thus, for your own convenience, you may want to define it in your LOGIN.COM file. See the *Introduction to VMS* for more information on LOGIN.COM files.

The Ada-from-CDD translator produces the following translation in the file INFO.ADA:

```
with SYSTEM; use SYSTEM;
with CDD_TYPES; use CDD_TYPES;
package CDD_TOP_MAIL_ORDER_INFO is

    -- CDD Path Name "CDD$TOP.MAIL_ORDER.INFO"

    type MAIL_ORDER_TYPE is
        record
            ORDER_NUM :    UNSIGNED_LONGWORD;  -- unsigned longword
            NAME :         STRING(1 .. 20);    -- text
            ADDRESS :      STRING(1 .. 20);    -- text
            CITY :         STRING(1 .. 19);    -- text
            STATE :        STRING(1 .. 2);     -- text
            ZIP_CODE :     STRING(1 .. 5);     -- text
            ITEM_NUM :     UNSIGNED_LONGWORD;  -- unsigned longword
            SHIPPING :     FLOAT;              -- F_floating
        end record;
```

```
        for MAIL_ORDER_TYPE use
            record
                ORDER_NUM           at 0    range 0 .. 31;
                NAME                at 4    range 0 .. 159;
                ADDRESS             at 24   range 0 .. 159;
                CITY                at 44   range 0 .. 151;
                STATE               at 63   range 0 .. 15;
                ZIP_CODE            at 65   range 0 .. 39;
                ITEM_NUM            at 70   range 0 .. 31;
                SHIPPING            at 74   range 0 .. 31;
            end record;

        for MAIL_ORDER_TYPE'SIZE use 624;

end CDD_TOP_MAIL_ORDER_INFO;
```

You can then use this package in an Ada program as you would use of any other Ada package. For example:

```
with CDD_TOP_MAIL_ORDER_INFO; use CDD_TOP_MAIL_ORDER_INFO;
procedure USE_MAIL_DATABASE is
begin

   -- Work with the mail database using the type MAIL_ORDER_TYPE.

end USE_MAIL_DATABASE;
```

Chapter 8

Tasking

Ada tasks are entities that execute in parallel. For example, you can use tasks to take data concurrently from several sources, you can use them to do terminal input-output and a series of calculations at the same time, or you can use them to call asynchronous VMS system services.

This chapter provides information on how to use VAX Ada tasks effectively, giving, in particular, information on how to use tasks in the VMS environment.

If you are not familiar with Ada tasking, read Chapter 9 of the *VAX Ada Language Reference Manual* before reading this chapter. For information on the VMS concepts presented in this chapter, see the *Introduction to VMS System Services* and the *Introduction to the VMS Run-Time Library*.

For information on the interaction of tasks with VAX Ada input-output facilities and exception handling, see Chapters 3 and 4.

8.1 Introduction to Using Ada Tasks on the VMS Operating System

A *task* is an entity whose execution proceeds in parallel with the execution of other tasks. The Ada language allows you to declare both task types and task objects.

A special kind of task—an *environment task*—is automatically created when you run a main VAX Ada program. This task—the *main task*—first elaborates any library packages associated with the program, and then calls the main program (see Chapter 10 of the *VAX Ada Language Reference Manual*). When execution of the main program is completed, and all tasks that depend on its library packages terminate, the main task is deleted, and control returns to the VMS operating system.

Any task is said to *depend* on a number of *masters*. Blocks, tasks, subprograms, or library packages can all be the master of a task. If you declare a task object in a block, for example, the block is the master of the created task, and the task depends on the block. If the block is executed within the statement part of a subprogram, then the subprogram is another master of the task and the task depends on it too. An *immediate master* is the master that immediately contains the declaration of a task object, or that immediately contains the definition of the access type whose designated type is a task type. A key rule is that control cannot leave a master until all of its dependent tasks have terminated. Thus, if some dependent task chooses not to terminate, none of its masters can exit, and the program, or a portion of it, appears to "hang."

Each time you create a task (for example, by declaring a task object, or evaluating an allocator that points to a task object), VAX Ada automatically creates a *task control block* to manage the task. When the task is activated, VAX Ada creates a stack to be used by the statements that the task will execute, and allows the task to compete for the processor on which your process is executing.

Because all tasks in any Ada program (including the main task) run in the context of a single process, control can switch from one task to another very quickly. This switch can occur at or during any of the several machine instructions that make up an Ada program statement; that is, the switch can occur midway through the execution of an Ada source line. Because of task switching, you will often need to synchronize the execution of tasks in your program to get the behavior you desire. Synchronization involves making sure that the right things happen in the right order. The usual means of synchronizing tasks is to use Ada's rendezvous mechanism.

Example 8–1 shows the use of tasks to do input-output and another activity in parallel. The example program reads in an array of integers, and sorts them using a quick sort. The sorting is done by one task, and another task (running in parallel) allows you to see how the sort is progressing by executing input-output statements while the sort is being done. The comments in the example point out various tasking concepts (activation, synchronization, and so on). These concepts are defined fully in the *VAX Ada Language Reference Manual*.

Example 8–1: Interactive Array Sort Using Tasks

```
-- This example shows that one task can execute while another
-- waits for input-output.
--
-- The main program has a background task that sorts an array while
-- another task interacts with the terminal user. The interactive
-- task, upon user command, will display the array at any time
-- during the sort.
--
-- Before running this program, make sure that the input
-- file is not a process-permanent file (SYS$INPUT); otherwise,
-- the lower-priority sorting task will not run. To avoid
-- using SYS$INPUT, first type the following:
--
-- $ DEFINE ADA$INPUT TT
-------------------------------------------------------------------

-- Program to sort an array by means of a quick sort and examine
-- it as it is sorted.
--
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure TASKSORT is

   -- Enable time slicing.
   --
   pragma TIME_SLICE(0.3);

   type QUICKARRAY is array (INTEGER range <>) of INTEGER;

   -- Array to be sorted and shared among tasks.
   --
   A     : QUICKARRAY(1..120);
   ASIZE : INTEGER;

   -- Force array references to be made to actual
   -- array storage (rather than to a copy).
   --
   pragma VOLATILE(A);

   SENTINEL : STRING(1..120) := (1..120 => ' ');

   -- Task to synchronize access to the array being sorted.
   --
   task GRANTOR is
      entry GRAB_ACCESS;
      entry RELEASE_ACCESS;
   end GRANTOR;
```

(continued on next page)

Example 8–1 (Cont.): Interactive Array Sort Using Tasks

```
-- Lower priority background task to do sorting.
--
task QUICK is
   entry QSORT (ARG_I, ARG_J: INTEGER);
   pragma PRIORITY(3);
end QUICK;

-- Higher priority interactive task to display
-- sort results.
--
task USER is
   pragma PRIORITY(7);
end USER;

task body GRANTOR is
begin
   loop
      select
         accept GRAB_ACCESS;
         accept RELEASE_ACCESS;
      or
         terminate;
      end select;
   end loop;
end GRANTOR;

task body QUICK is
   I, J, MIDDLE_KEY : INTEGER;
   KEY_INDEX        : INTEGER;
   KEY              : INTEGER;

   function FIND_MIDDLE (I,J: INTEGER) return INTEGER is
      FIRST : INTEGER;
      KEY   : INTEGER;
   begin
      FIRST := A(I);
      for KEY in (I + 1)..J loop
         if A(KEY) > FIRST then
            return KEY;
         elsif A(KEY) < FIRST then
            return I;
         end if;
       end loop;
       return 0;
   end FIND_MIDDLE;
```

(continued on next page)

Example 8–1 (Cont.): Interactive Array Sort Using Tasks

```
function DIVIDE_ARRAY (I,J          : INTEGER;
                       MIDDLE_KEY : INTEGER)
   return INTEGER is
   LEFT, RIGHT, TEMP : INTEGER;
begin
   --
   -- Rendezvous to synchronize access to the
   -- array for partitioning.
   --
   GRANTOR.GRAB_ACCESS;
   LEFT  := I;
   RIGHT := J;
   loop
      TEMP := A(LEFT);
      A(LEFT) := A(RIGHT);
      A(RIGHT) := TEMP;
      while A(LEFT) < MIDDLE_KEY
         loop
            LEFT := LEFT + 1;
         end loop;
      while A(RIGHT) >= MIDDLE_KEY
         loop
            RIGHT := RIGHT - 1;
         end loop;
      exit when LEFT > RIGHT;
   end loop;

   -- Rendezvous to synchronize end of
   -- array access.
   --
   GRANTOR.RELEASE_ACCESS;
   PUT_LINE("Partial sort complete.");
   return LEFT;
end DIVIDE_ARRAY;

procedure QUICK_SORT (I,J: INTEGER) is
begin
   KEY_INDEX := FIND_MIDDLE(I,J);
   if KEY_INDEX /= 0 then
      delay 8.0;
      MIDDLE_KEY := A(KEY_INDEX);
      KEY := DIVIDE_ARRAY(I,J,MIDDLE_KEY);
      QUICK_SORT(I,KEY-1);
      QUICK_SORT(KEY,J);
   end if;
end QUICK_SORT;
```

(continued on next page)

Example 8–1 (Cont.): Interactive Array Sort Using Tasks

```
begin
   select
      accept QSORT (ARG_I,ARG_J: INTEGER) do
         I := ARG_I;
         J := ARG_J;
      end QSORT;
   or
      terminate;
   end select;
   PUT_LINE("The sorting task has started.");
   QUICK_SORT(I,J);
   PUT_LINE("The sorting task has completed.");
end QUICK;

procedure PRINT_ARRAY is
begin
   --
   -- Again, use GRANTOR task rendezvous to
   -- synchronize array access for printing.
   --
   GRANTOR.GRAB_ACCESS;
   for I in 1..ASIZE
      loop
         PUT(A(I),WIDTH=>3);
      end loop;
   NEW_LINE;
   GRANTOR.RELEASE_ACCESS;
end PRINT_ARRAY;

task body USER is
   I    : INTEGER;
   LAST : NATURAL;
begin
   PUT_LINE("Type in the number of " &
            "integers you want sorted,");
   PUT_LINE("and then press RETURN.");
   GET(ASIZE);
   PUT_LINE("Now, type in a string of integers, " &
            "separated by spaces, ");
   PUT_LINE("that you want sorted. End the " &
            "string with a RETURN.");
   for I in 1..ASIZE
      loop
         GET(A(I));
      end loop;
   PUT("The initial array is ");
   PRINT_ARRAY;
```

(continued on next page)

Example 8–1 (Cont.): Interactive Array Sort Using Tasks

```
      -- Start the sorting task.
      --
      QUICK.QSORT(1,ASIZE);

      -- Allow the terminal user to see the array at any time.
      --
      loop
         PUT_LINE("Press RETURN to see partially " &
                  "sorted array or e to exit.");
         GET_LINE(SENTINEL, LAST);
         if LAST >= 1 and then (SENTINEL(1) = 'E' or else
                                SENTINEL(1) = 'e') then
            exit;
         end if;
         PRINT_ARRAY;
      end loop;
      exception
         when END_ERROR =>
            PUT_LINE("That's all folks!");
         when others =>
            PUT_LINE("You've made a mistake; try again.");
            SKIP_LINE;
            TASKSORT;  -- Re-call main program.
   end USER;

begin -- Activate all tasks (GRANTOR, QUICK, USER);
      -- all tasks depend on the environment task created
      -- for the main program TASKSORT.
    null;
end TASKSORT;
```

8.2 Task Storage Allocation

Each task created in your program requires a certain amount of storage: when your program creates a task, a task control block is allocated; when the task is activated, a stack is allocated. Because the VMS operating system is a virtual memory operating system, the number of tasks you can create is limited only by the amount of virtual storage available to your process.

The following sections discuss task storage allocation, and explain how you can control it and how it can be important in a mixed-language environment.

8.2.1 Storage Created for a Task Object—The Task Control Block

When your program creates a task object, VAX Ada allocates a block of storage—a *task control block*—to keep track of that task's execution. For example, a task control block is allocated for each of the following task objects:

```
task type MY_TASK;
T : MY_TASK;
```

```
task type MY_TASK;
type MY_TASK_POINTER is access MY_TASK;
PT : MY_TASK_POINTER;
 . . .
PT := new MY_TASK;
```

The task control block is deallocated when control leaves the immediate master (another task or a currently executing block or subprogram) upon which the task depends—not when the task terminates. See Section 8.1 of this manual and Chapter 9 of the *VAX Ada Language Reference Manual* for a definition of masters and dependence.

The size of a task control block depends on the characteristics of the task's type. In other words, its size increases in proportion to the number of single entries in the task type, the total number of *members* of all of its entry families, and the number of single entries that have been specified to receive ASTs (see Section 8.6). In particular, if you specify an entry family with a very large discrete range, a large amount of storage is allocated when a task of the type is created; to maximize execution speed, an entry-call queue is allocated for each member of an entry family. For example, the following declaration will cause a large amount of storage to be allocated:

```
entry X (1..100_000);
```

You can estimate the number of pages (512 bytes per page) to be allocated for a task control block as follows; remember that you need to round up fractional results to a whole number:

$$TCB_SIZE\ (pages) = \frac{FIXED_AMOUNT\ +\ (E*12.2)\ +\ (AST_E*28)}{512}$$

where

FIXED_AMOUNT = 3000 bytes

E = the number of single entries plus the number of members in all entry families

AST_E = the number of single entries that have been specified to receive ASTs; that is, those entries declared with the pragma AST_ENTRY

For most task types (that is, those having fewer than a few hundred total entries), the storage consumed by the task control block is relatively small. Note that the main task has no entries, so the main task control block has a constant size.

You can reduce the size of the task control block by reducing the number of entries, the number of entry family members, and the number of entries that have been declared with the pragma AST_ENTRY. In addition, you can cause the storage consumed by a task control block of a terminated task to be released by arranging for control to leave its immediate master; see Example 8–2.

Note from Example 8–2 that storage for only one task control block is consumed at any one time, even though 100,000 tasks are created. This is because the block is the immediate master of tasks declared to be of type ACCESS_TO_TASK. If X were instead declared to be of type OUTER_ ACCESS, storage for 100,000 task control blocks would need to be allocated (even though all blocks but one are terminated), and the exception STORAGE_ERROR may be raised.

Thus, as your program creates and terminates many tasks, storage for terminated tasks will accumulate. To reduce the accumulated storage for terminated tasks, you should arrange the program so that the immediate master on which the task depends is as innermost as feasible. This strategy, however, saves space at the expense of more execution time. To minimize execution time, follow this (opposing) strategy: arrange the program so that task types and task declarations are as outermost as feasible to minimize the number of tasks that are created and terminated.

8.2.2 Storage Created for a Task Activation—The Task Stack

Each time a task is activated, a *task stack* is allocated. The storage for the task stack is deallocated as soon as the task is terminated.

Example 8–2: Leaving a Master to Release a Task Control Block

```
procedure RELEASE is

   task type SOME_TASK;

   type OUTER_ACCESS is access SOME_TASK;

   task body SOME_TASK is
   begin
      delay 0.5;              -- Simulate doing some useful work.
   end SOME_TASK;

begin

   -- This loop creates 100,000 tasks.  X is assigned
   -- to refer to each new task in turn.  Each task
   -- terminates a short time after creation.
   --
   for I in 1..100000 loop
      declare
         type ACCESS_TO_TASK is access SOME_TASK;
         X : ACCESS_TO_TASK;
      begin
         X := new SOME_TASK;
         delay 1.0;  -- Wait long enough to be sure that
                     -- X is terminated.
      end;           -- Await termination of all tasks
                     -- referred to by type ACCESS_TO_TASK,
                     -- and free their storage.
   end loop;

end RELEASE;         -- Await termination of all tasks
                     -- referred to by type OUTER_ACCESS,
                     -- and free their storage.
```

In the absence of the VAX Ada pragma MAIN_STORAGE, the stack for a main task is allocated in the P1 region of the process in which the program is running. When it is so allocated, the stack of the main task has no definite limit; as long as your process has not used up all of its virtual memory, the main task stack is automatically expanded as needed.

The storage for all other tasks, including a main task declared with the pragma MAIN_STORAGE, is allocated in the P0 region. The stacks for these tasks are fixed in size, and are not expanded.

The task stack allocated for any VAX Ada task (including the main task) has two areas: a *working storage area* and a *top guard area*. The working storage area is used during normal task execution for the storing of variables, call frames, and so on. The top guard area is a set of pages (512 bytes per page) at the top of the stack. These pages are inaccessible to your

program—that is, attempts to read or write them will cause a hardware access violation (SS$_ACCVIO), which will usually terminate your program immediately. The purpose of these guard pages is to help you detect accidental overflow of the working area of the stack. For example, accidental stack overflow can occur as follows:

- When a task with a fixed-size stack executes non-Ada code for which stack checking is not performed (see Section 8.2.3)
- When storage size checks are suppressed when you compile the program (see Section 4.3)

NOTE

AST routines execute on the stack of whatever task is currently active. See Section 8.6 for more information on AST routines and tasks.

Unless you specify otherwise, the sizes of the working area and the top guard area of all task stacks are set by the VAX Ada run-time library. For tasks with fixed-size stacks, the working area is set by default to 60 pages, and the top guard area is set to 10 pages. The default stack allocation for tasks with fixed-size stacks allows an additional 10 pages of stack for calls to non-Ada routines, which is adequate for most routines, including VMS system service and Run-Time Library routines.

You may need to specify the sizes of a task's stack areas for a number of reasons:

- You may find that a task is raising the exception STORAGE_ERROR, and you want to increase its working area.
- You may find that a task does not need all of its default stack allocation, and you may wish to reduce the working area so that the unused storage can be put to other use by your program (for example, if your program creates many tasks).
- You have not called any non-Ada routines, and you are not having any stack overflow (you have not suppressed checks and STORAGE_ERROR is not being raised). You may thus wish to decrease the top guard area and put the storage to other use.
- You may suspect that some non-Ada routine might be overflowing the stack, and you may wish to increase the top guard area in an attempt to detect the overflow.
- In the case of a main task, you may wish to emulate the behavior of tasks on a VAX system running the VAXELN executive (see the *VAXELN Ada User's Manual* for more information on VAXELN Ada).

To control the storage allocated for a main task stack (and to force a fixed-size, P0 space stack allocation), use the pragma MAIN_STORAGE; to control the storage allocated for all other task stacks, use the STORAGE_SIZE length representation clause attribute and the VAX Ada pragma TASK_STORAGE. The following sections describe how to control task stack storage.

8.2.2.1 Controlling the Stack Sizes of Task Objects

To control the working storage area of the stack of a task object, you can apply a representation clause to the type used to declare that object. For example, the following length clause sets the working storage size for the task type NEEDS_BIG_STACK to 300 pages:

```
for NEEDS_BIG_STACK'STORAGE_SIZE use 300*512;
```

Any task objects of this type will have 300-page working storage areas.

Note that if you specify a size of zero (bytes) with 'STORAGE_SIZE, a default stack size is used. Also, regardless of the size you specify, at least 21 pages of additional space are allocated for task management purposes. In any case, the VMS Debugger can help you to determine and tune the amount of storage you need for a stack working area; see *Developing Ada Programs on VMS Systems*.

To control the top guard area of a task object, you can use the VAX Ada pragma TASK_STORAGE to set the amount of guard storage allocated for the task type used to declare that object. For example, the following statement sets the top guard area of the task type NEEDS_BIG_STACK to zero:

```
pragma TASK_STORAGE(NEEDS_BIG_STACK, 0)
```

Any object of this type will have a default working storage size (unless a representation clause was also specified) and no guard area.

Note from the *VAX Ada Language Reference Manual* that 'STORAGE_SIZE and the pragma TASK_STORAGE apply only to task *types*; to apply them to a single task, you must convert the single task to a task type declaration and then a task object declaration. You should use task types when you begin coding your tasking programs, if you anticipate using representation clauses or the pragma TASK_STORAGE later on; you may have to rewrite your program if you have single tasks that are later declarations and that need to be converted to task types (a task type declaration cannot be a later declaration). See Chapter 3 of the *VAX Ada Language Reference Manual* for more information on later declarations; see Chapter 13 of the *VAX Ada*

Language Reference Manual for a description of the syntax and rules for using 'STORAGE_SIZE and the pragma TASK_STORAGE.

Example 8–3 shows the control of stack areas using 'STORAGE_SIZE. and the pragma TASK_STORAGE.

Example 8–3: Controlling the Size of a Task's Stack

```
procedure CONTROL is

   task type NEEDS_BIG_STACK;

   -- Set the stack working area of tasks of type NEEDS_BIG_STACK
   -- so that these tasks can handle the deepest call of the
   -- recursive procedure CALL_SELF. (The value 76 is sufficient
   -- storage for one activation of the procedure CALL_SELF.)
   --
   for NEEDS_BIG_STACK'STORAGE_SIZE use 30000*76;

   -- Decrease the top guard area of the stack to 0 because the
   -- task NEEDS_BIG_STACK does not call outside Ada. Thus,
   -- no guard pages are needed.
   --
   pragma TASK_STORAGE(NEEDS_BIG_STACK, 0);

   T : NEEDS_BIG_STACK;

   task body NEEDS_BIG_STACK is
      procedure CALL_SELF (I : INTEGER) is
      begin
         if I < 30000 then
            CALL_SELF(I + 1);
         end if;
      end CALL_SELF;
   begin
       CALL_SELF(1);
   end NEEDS_BIG_STACK;

begin
   null;
end CONTROL;
```

8.2.2.2 Controlling the Size of a Main Task Stack

Main task stacks usually have no definite limit, and are automatically expanded as needed in the VMS environment. However, VAX Ada provides the pragma MAIN_STORAGE to allow you to control the size (and the allocation space) of a main task stack. As noted in previous sections, this pragma causes the size of the main task stack to be fixed; it also causes the stack to be allocated in the P0 region rather than in the P1 region. This

pragma is intended primarily to allow control over the sizing of main task stacks in a VAXELN environment (with VAXELN Ada); thus, it is generally useful in the VMS environment when you need to simulate the behavior of a VAXELN main task when you are working with VAX Ada and a VMS target. See the *VAXELN Ada User's Manual* for more information on VAXELN Ada.

The pragma MAIN_STORAGE has two parameters, WORKING_STORAGE and TOP_GUARD, which allow you to specify (in bytes) either or both the working storage and top guard areas of the main task. For example:

```
procedure MAIN_PROGRAM is
   pragma MAIN_STORAGE (WORKING_STORAGE => 100*512,
                        TOP_GUARD       => 0);
begin
    . . .
end;
```

Here, the working storage area of the main program is limited to 100*512 bytes (that is, 100 pages), and the top guard area is set to zero. If you specify WORKING_STORAGE or TOP_GUARD alone, a default value is chosen for the omitted parameter. A default stack size is also used if you specify a value of 0 for WORKING_STORAGE; regardless of the value specified for WORKING_STORAGE, at least three pages of additional space are allocated for task management purposes. In any case, the VMS Debugger can help you to determine and tune the amount of storage you need for a stack working area; see *Developing Ada Programs on VMS Systems*.

See Chapter 13 of the *VAX Ada Language Reference Manual* for a description of the syntax and rules for using the pragma MAIN_STORAGE.

8.2.3 Stack Overflow and Non-Ada Code

You are protected from stack overflow in an Ada program because VAX Ada raises the exception STORAGE_ERROR when an attempt is made to overflow either the main stack or an Ada task stack. In addition, the default 10-page stack storage allocated for each non-Ada call should be adequate protection against stack overflow for most non-Ada routine calls (see Section 8.2.2).

However, be aware that non-Ada routines, VMS system services, Run-Time Library routines, and so on do not check for stack overflow. Thus, when you call a non-Ada routine from an Ada program, it may be possible that the stack of the main task or an individual task will overflow, and the Ada program will not be able to detect it because the exception STORAGE_ERROR will not be raised. Such an undetected stack overflow could result

in random changes to various locations beyond the storage allocated for the stack. Because the correct operation of the Ada program may depend on such locations, undetected stack overflow could make your program erroneous.

Thus to be safe, do not mix Ada and non-Ada programs without checking for stack overflow. You can use the top guard areas of tasks in your program to detect if a non-Ada routine causes the stack to overflow (see Section 8.2.2 for information about the top guard area). If you make the size of the guard pages in the top guard area large enough, then undetected overflows that are not larger than the guard pages will raise a hardware access violation (SS$_ACCVIO) exception, which will usually terminate your image immediately.

The VMS Debugger can be of great help in detecting stack overflow. The debugger will perform an automatic stack check for you, and can display the amount of stack space in use in any task. For further information, see *Developing Ada Programs on VMS Systems*.

8.3 Task Switching and Scheduling

VAX Ada implements the Ada language requirement that when two tasks are eligible for execution, and they have different priorities, the lower priority task will not execute while the higher task is waiting. Thus, the VAX Ada run-time library keeps a task running until either the task is suspended or a higher priority task becomes ready.

NOTE

> The term "suspend" is used in this chapter (as it is used in the *VAX Ada Language Reference Manual*) to mean that execution of the task is temporarily stopped—the task is waiting for another event, such as the acceptance of an entry call, to occur before execution resumes. "Suspend" does not refer to the VMS system service $SUSPND.

The default VAX Ada task scheduling may be described as "first-in-first-out (FIFO), with preemption." This phrase means that tasks of equal priority are processed in first-in-first-out order: a task is run until it suspends; when it later resumes, it is placed at the rear of the ready queue for its priority level. The term "with preemption" means that VAX Ada will preempt a running task if a higher priority task becomes ready; this behavior is required by Ada rules (see Section 9.8 of the *VAX Ada Language Reference Manual*). The preempted task is placed at the front of the ready queue for its priority level. Then, when the higher priority task suspends, the

preempted task resumes execution. In other words, the preemption of a lower priority task does not imply any cycling of the ready queue for that priority.

This scheduling strategy has the benefit that the execution of lower priority tasks is minimally affected by any change in the exact instant at which the higher priority task becomes ready (which can change from run to run). In other words, this scheduling method increases program repeatability, and helps you debug your program.

However, FIFO scheduling is not necessarily fair to tasks of equal priority that are eligible for execution but that are not yet running. These waiting tasks can exhibit sluggish response times, especially if they are interacting with a terminal. In fact, they will *never* get to run if the running task does not become suspended. FIFO scheduling permits a running task to capture the processor.

There are a number of ways in which you can control scheduling. You can use the pragma PRIORITY to give the more important tasks higher priorities, and thus increase their responsiveness. The range of possible VAX Ada task priorities is from 0 to 15; in the absence of this pragma, VAX Ada tasks have a default midrange priority of 7. (Note that task priority has no effect on the priority of your process; from the VMS operating system's point of view, the process priority applies to the execution of every task in your program.) For example, the following statements set the priority of the task IMPORTANT_TASK to 14:

```
task IMPORTANT_TASK is
   pragma PRIORITY(14);
end IMPORTANT_TASK;
```

This pragma can appear only in a task specification or in the outermost declarative part of a main subprogram. See Chapter 9 of the *VAX Ada Language Reference Manual* for a description of the syntax and use of the pragma PRIORITY.

If, instead, you want to increase the fairness of the scheduling by limiting the execution time for any particular task, you can specify a time slice with the VAX Ada pragma TIME_SLICE or with the procedure SYSTEM_RUNTIME_TUNING.SET_TIME_SLICE. By specifying a time slice, you change the default FIFO scheduling strategy to what is known as *round-robin scheduling*. Round-robin scheduling causes tasks of the same priority to take turns at the processor, thus preventing a nonsuspending task from capturing the processor.

Time slicing is useful during development to help you find race conditions and deadlocks. It tends to make tasks of equal priority execute in an arbitrary order, and thus stresses the tasking logic in your program. However, consider removing the pragma in production code to both reduce overhead and possibly enhance the reliability of your program. (In some applications, time slicing during production may enhance reliability—or may be required—if some tasks have code paths where they fail to suspend.)

You specify a time slice with the pragma TIME_SLICE as follows:

```
pragma TIME_SLICE(static-expression);
```

The static expression must be of the type SYSTEM.DURATION. A value of 0.0 (the default) or less disables time slicing. This pragma has an effect only if it appears in the outermost declarative part of a main program; see Chapter 9 of the *VAX Ada Language Reference Manual* for a complete description.

See the specification of the package SYSTEM_RUNTIME_TUNING in Appendix B for information on using the SET_TIME_SLICE procedure.

If you specify a time slice, you must realize that, while you are increasing fairness, you are paying a price in terms of increased task switching overhead (the overhead increases for smaller time-slice values) and more difficult debugging. Time-slice values below 0.01 second do not result in faster time slicing because the smallest time increment supported by the VMS operating system is 0.01 second.

8.4 Special Tasking Considerations

Use of tasks in an Ada program requires some care, because, like any other language construct, tasking has its own characteristic set of programming pitfalls. (Infinite looping, for example, is a characteristic pitfall of while loops.)

The following topics are discussed in this section: deadlock, busy waiting, tentative rendezvous, delay statements, abort statements, interrupting program execution with CTRL/Y, shared variables, and reentrancy.

8.4.1 Deadlock

Deadlock is a condition in which each task in a group of tasks is suspended and no task in the group can resume its execution until some other task in the group executes. Deadlock is also called "circular wait."

The possibility that Ada tasks may deadlock is a property of the Ada language. You can eliminate deadlock with careful program design. In addition, the VMS Debugger provides special task debugging commands that can help you detect deadlocks; see *Developing Ada Programs on VMS Systems*.

The following examples show some of the more common forms of Ada deadlock: exception-induced, self-calling, circular-calling, and dynamic-circular-calling.

An *exception-induced deadlock* occurs when an exception prevents a task from answering one of its entry calls; if the exception had not occurred, there would be no deadlock. This kind of deadlock occurs when an unhandled exception in an Ada task must wait for the termination of local dependent tasks before propagating. Exception-induced deadlock is more subtle than the other kinds of deadlock because, were it not for the exception, the program would be deadlock free. Example 8–4 shows an exception-induced deadlock.

A *self-calling deadlock* occurs when a task calls one of its own entries. The call cannot be completed until the call is answered, and the call cannot be answered because the task itself becomes suspended at the call. Self-calling deadlock becomes more subtle if the task calls a procedure that calls the task. Example 8–5 shows self-calling deadlock.

Example 8–4: An Exception-Induced Deadlock

```
procedure EXCEPTION_INDUCED is

   task PARENT is
      entry E;
   end PARENT;

   task body PARENT is
   begin
      declare
         task CHILD;

         UNANTICIPATED_EXCEPTION : exception;

         task body CHILD is    -- Exceptions wait for any
         begin                 -- task declared within a
            PARENT.E;          -- unit declared within a task.
         end;

      begin
         raise UNANTICIPATED_EXCEPTION;  -- Exception occurs
         accept E;                       -- here; CHILD's call
                                         -- never accepted.

      end;                               -- Parent waits here
                                         -- for termination
                                         -- of CHILD.
   end PARENT;

begin
    null;
end EXCEPTION_INDUCED;
```

Example 8–5: A Self-Calling Deadlock

```
procedure SELF_CALL is

   task type T is
      entry E;
   end T;

   Y : T;

   procedure P(X : T) is    -- Calls entry E in task X.
   begin
      X.E;
   end P;

   task body T is
   begin
      P(Y);                 -- Never returns.
      accept E;
   end T;

begin
   null;
end SELF_CALL;
```

A *circular-calling deadlock* occurs when a task calls another task that calls another task, and so on, and the last task calls the first task. One way you can eliminate circular-calling deadlock is by restricting your program so that task calls form a strict hierarchy. Example 8–6 shows circular-calling deadlock.

Example 8–6: A Circular-Calling Deadlock

```
procedure CIRCULAR_CALL is

   task type T1 is
      entry E;
   end T1;

   task type T2 is
      entry E;
   end T2;

   Y : T1;
   Z : T2;

   procedure P is
   begin
      Z.E;
   end P;

   task body T1 is
   begin
      P;
   end T1;

   task body T2 is
   begin
      Y.E;
   end T2;

begin
   null;
end CIRCULAR_CALL;
```

A *dynamic-circular-calling deadlock* occurs when a series of entry calls forms a circle as in either of the previous two cases, but at least one of the calls is a timed or conditional entry call in a loop that completes only if the rendezvous occurs. Thus, with dynamic-circular-calling deadlock, at least one task is executing, but no progress can be made. Example 8–7 shows a dynamic-circular-calling deadlock.

Example 8–7: A Dynamic-Circular-Calling Deadlock

```
procedure DYNAMIC_CALL is

   task type T is
      entry E;
   end T;

   Y : T;

   procedure P(X : T) is
      DONE : BOOLEAN := FALSE;
   begin
      while not DONE loop
         select
            X.E;
            DONE := TRUE;
         or
            delay 0.5;          -- This alternative is always
                                -- chosen.
         end select;
      end loop;
   end P;

   task body T is
   begin
      P(Y);                     -- The call to P never returns.
      accept E;
   end T;

begin
   null;
end DYNAMIC_CALL;
```

8.4.2 Busy Waiting and Non-Ada Code

Busy waiting is a programming technique that repeatedly tests a variable, sometimes called a "flag" or a "spin lock," to determine if some event has occurred. When the event does occur, another instruction sequence is presumed to execute and set the flag, thereby ending the looping.

Busy waiting is sometimes desirable when an event will occur quickly, and it is justifiable to use CPU time to wait for it. It is also desirable when no other suitable synchronization methods (such as rendezvous) are available.

Busy waiting has some undesirable characteristics, however. First, assumptions about an event that were true when the code was written may no longer be true when the code is executed; as a result, a large, unanticipated

amount of CPU time may be consumed at execution time. For a process running under the VMS operating system, this usually means that the process is using processor resources that another process could use to advantage.

Second, when tasks execute busy-waiting code, the effect can be unpredictable. Consider the following situation:

- One task is executing a wait loop, while another task is expected to set the flag.
- The task executing the busy-waiting code has the highest priority in the program.

If time slicing is not enabled, deadlock will develop because the busy-waiting task does not suspend and no other task (including the flag-setting task) can be scheduled (see Section 8.3 for a discussion of first-in first-out scheduling). (This behavior is in accordance with Ada rules.) The situation can be improved only slightly if time slicing is enabled. The deadlock can still develop if the flag is to be set by a task of lower priority (even with time slicing, a low-priority task cannot be scheduled while a higher priority task is ready).

Because of these potential problems, avoid busy waiting. VAX Ada does not use busy waiting, so if your program uses only VAX Ada, you should not encounter this kind of deadlock.

If you do discover that your tasking program is caught in a busy waiting loop by some software over which you have no control, you can probably correct the problem by setting all of your task priorities to the same value (or, equivalently, by eliminating all specifications of the pragma PRIORITY) and by enabling time slicing with the pragma TIME_SLICE (see Section 8.3) or with the procedure SYSTEM_RUNTIME_TUNING.SET_TIME_SLICE (see Appendix B.).

8.4.3 Tentative Rendezvous

Ada provides a number of "tentative" rendezvous constructs: conditional entry calls, select-with-else combinations, and even timed entry call and select-with-delay combinations.

These constructs are most often coded in loops. They have the potential effect of causing the task executing such loops to take over the processor if the task has the same or a greater priority as all of the other tasks available for execution. Then, if the executing task does take over the processor, it could end up executing indefinitely if it depends on any of the tasks it is

preventing from executing. Thus, tentative rendezvous constructs require special care; think of them as forms of busy waiting.

8.4.4 Using Delay Statements

VAX Ada implements the delay statement as a call to the VMS system service SYS$SETIMR. Thus, each delay statement places an entry in the system timer queue, which, in turn, affects the VMS operating system Timer Queue Entry Limit (TQELM) quota. Each delay statement also makes use of the SYS$SETIMR routine's ASTADR parameter, which specifies an AST routine. Thus, the use of delay statements can also affect (or possibly exceed) the AST Queue Limit (ASTLM) quota.

In effect, the TQELM quota limits the number of concurrent Ada delay statements: when a request is made that would cause the TQELM quota to be exceeded, the call to SYS$SETIMR stalls until a timer entry packet becomes available. In other words, the call stalls until an active delay expires, and the delay will not start until the call is made. A low quota can affect any Ada statement containing the reserved word **delay**; it can also affect the duration of a time slice, if the main program uses the pragma TIME_SLICE.

You can eliminate this delay anomaly by increasing the TQELM quota for your process. The TQELM quota should exceed the number of simultaneous statements involving delay that can be in progress at one time (an upper bound is the peak number of tasks that can exist simultaneously in your program). One additional timer entry is required if your program uses the pragma TIME_SLICE. You may need to increase the TQELM quota further if your program executes any other timer-related system services.

To increase the TQELM or ASTLM quota for your process, see your system manager. The *Guide to Maintaining a VMS System* gives details on how to adjust these quotas.

8.4.5 Using Abort Statements

Be careful when you use abort statements: an abort statement can terminate a task when it should not be terminated, and thus can lead to erroneous execution. You should use abort statements only when you require unconditional termination, and only when you are sure that it is safe to do so. For example, if you abort a task with an asynchronous system service request in progress (such as SYS$QIO), the task can become terminated and its stack storage reallocated to some other use before the VMS operating

system has written the result data. The result data could be written in some unexpected part of your program's data area.

VAX Ada implements the abort statement in a *synchronous* rather than an *asynchronous* form. An asynchronous implementation of the abort statement can cause completion of tasks at arbitrary points in their execution. The synchronous form causes tasks to become completed only at specific points in their execution (see Chapter 9 of the *VAX Ada Language Reference Manual* for a list of these points).

Synchronous abort has several benefits. One benefit is that when a task calls a non-Ada routine (and the routine does not result in a call to an Ada subprogram), the non-Ada routine will execute to completion even though the calling task has been aborted. Synchronous abort thus avoids problems that may result because non-Ada routines typically are not programmed to work correctly if they are only partially executed.

Unfortunately, synchronous abort also means that a task in an infinite loop cannot become completed unless it executes code that is a synchronization point for the abort statement. If you want to ensure that a task will become completed due to an abort statement in some section of code, you should insert a **delay** 0.0 statement there. The Ada language requires that an abnormal task become completed at a delay statement; thus, **delay** 0.0 is a fully transportable and low-overhead means of ensuring that completion can occur.

8.4.6 Interrupting Your Program with CTRL/Y

When you use CTRL/Y to interrupt the execution of an Ada program that contains tasks, you can expect some special side effects when you subsequently try to execute DCL commands. The DCL commands that you are most likely to enter after pressing CTRL/Y are: DEBUG, CONTINUE, EXIT, STOP, or a query such as DIRECTORY. For each of these commands (except CONTINUE), the current execution point of the process is modified by the VMS operating system, and execution resumes at the new location as follows:

- The CONTINUE command causes your program to begin execution at the same point at which the CTRL/Y interrupt occurred.
- The STOP command immediately terminates execution of your program, as well as terminating any tasks that may be active.
- The DEBUG command causes the VMS Debugger to be activated, and your process to continue its execution under control of the debugger.

- The EXIT command causes your program to execute the SYS$EXIT system service.
- Most other commands, like the DIRECTORY command, have the effect of first entering the EXIT command and then entering the command itself.

In addition, if a low-priority task is running when you press CTRL/Y, that task's priority affects the action taken. In particular, because a higher priority task may be scheduled almost immediately, the desired effect may not occur for a while, or might never occur. (The CONTINUE and STOP commands are not affected by the task's priority: the CONTINUE command because continuation makes the interruption irrelevant, and the STOP command because it does not resume execution in VAX user mode, where task switches take place.) For example, if you enter the DEBUG command, you may not enter the debugger immediately. If you enter the EXIT command, the process may continue execution and not exit. If you enter the DIRECTORY command, the result is equivalent to first entering the EXIT command and then the DIRECTORY command, so your process may continue executing.

There are two ways to control the results of using these commands. First, you can force your program to quit by entering the STOP command. When you do this, however, any established exit handlers will not have a chance to execute. For example, VAX Ada provides an exit handler for the input-output packages, and if you enter the STOP command to interrupt input-output, the VAX Ada handler will not have an opportunity to write the last partial record to whatever external files may be open at the time, to close those files, or to delete Ada temporary files.

A second solution is to use CTRL/C in conjunction with the VAX Ada predefined package CONTROL_C_INTERCEPTION, which allows you to run the debugger, exit the program, or enter a query command like the DIRECTORY command. The operations this package provides mimic the operations you can perform after pressing CTRL/Y. You invoke these operations by pressing CTRL/C anytime after the package has been elaborated. For example:

```
-- Enable CTRL/C interception prior to main
-- program execution (but not necessarily before
-- all library packages have been elaborated).
--
with CONTROL_C_INTERCEPTION;
pragma ELABORATE(CONTROL_C_INTERCEPTION);
procedure MY_MAIN_PROGRAM is
begin
 . . .
end MY_MAIN_PROGRAM;
```

The following example shows the response of this handler to CTRL/C:

```
Nothing can go wrong
go wrong
go wrong
go wrong
go wrong
 ^C
Ada CTRL/C Interceptor
Type: DEBUG, EXIT, CONTINUE, or a DCL command.
Ada_CTRL/C> DEBUG
DBG>
```

See Appendix B for the package specification of CONTROL_C_INTERCEPTION.

8.4.7 Using Shared Variables

The code generated by an Ada compiler may store the value of a variable in several, one, or no places in the memory of the machine (see Chapters 2 and 9). The compiler believes, unless instructed otherwise, that it can detect all attempts to read or write a variable, and arranges to have each of those attempts access the correct places.

The compiler makes assumptions based on the following rules:

- In the absence of a pragma SHARED or VOLATILE, a variable that is read by a task will not be written by another task until the reading task reaches a synchronization point.
- Also in the absence of a pragma SHARED or VOLATILE, a variable that is written by a task will not be read or written by another task until the writing task reaches a synchronization point. This rule is described more precisely in Section 9.11 of the *VAX Ada Language Reference Manual*.

These rules avoid the need for specifying either pragma SHARED or pragma VOLATILE for variables that are read or written by multiple tasks, provided that the reads and writes are implicitly or explicitly synchronized by tasking events such as a rendezvous.

If you want your program to read or write a variable in a way that does not satisfy these rules, then you must specify the pragma SHARED or VOLATILE for that variable. Otherwise, your program is erroneous.

The pragma SHARED is defined by the Ada language; the pragma VOLATILE is defined by VAX Ada.

Chapter 9 of the *VAX Ada Language Reference Manual* gives the Ada language assumptions about shared variables and gives the usage rules and syntax for the pragma SHARED. The syntax is given here for convenience:

```
pragma SHARED (variable_simple_name);
```

The named variable must be declared by an object declaration and must be of a scalar or access type.

Chapter 9 of the *VAX Ada Language Reference Manual* also gives the usage rules and syntax for the pragma VOLATILE. The syntax is given here for convenience:

```
pragma VOLATILE (variable_simple_name);
```

The named variable must be declared by an object declaration but can be of any type.

The pragma SHARED, when applied to a variable, tells the compiler that any write to that variable must be made visible to reads by other tasks immediately, not just when the current task reaches a synchronization point.

The pragma SHARED also tells the compiler that two successive reads or a write followed by a read may return two different values, even though there is no intervening synchronization point. Furthermore, the pragma SHARED tells the compiler that it may have to generate special code to guarantee that complete values, not half the bit pattern of an old value and half the bit pattern of the new value, are read.

In implementing the pragma SHARED, VAX Ada guarantees that every read or update of a shared variable is a synchronization point. VAX Ada accomplishes this by ensuring the following actions for updates:

- When a shared variable is updated, the value is written to the storage allocated for the variable.
- Each write is performed as an indivisible operation (to exclude the possibility of another task reading a partially updated value).
- An interlocked instruction is executed so that all VAX processors that share memory with the current processor are informed that the update has taken place (to keep other processors from continuing to read an old value for the variable and for any volatile variables out of their memory cache).

Similarly, VAX Ada ensures the following actions for reads:

- Each read is from the storage allocated for the shared variable.
- Each read is performed as an indivisible operation; however, other processors are not informed of the read.

VAX Ada ensures the indivisibility of reads and updates of variables specified by a pragma SHARED as follows:

- Only those scalar or access variables whose storage size does not exceed a longword (32 bits) are allowed. For example, you cannot specify variables of the type D_FLOAT, G_FLOAT, or H_FLOAT in a pragma SHARED.
- By allocating longword-aligned longwords for all shared variables whose storage size is larger than a byte.
- By using a restricted set of VAX instructions to read and write such variables.

The pragma VOLATILE tells the compiler that any write by the current task to the specified variable must be made visible to reads by other tasks before the current task writes a variable for which the pragma SHARED was specified, not just when the current task reaches a synchronization point. The pragma VOLATILE does not guarantee that the change will be seen by another task before then. The compiler must make such writes immediately visible to ASTs and system services that are invoked by the task and read the variable.

The pragma VOLATILE also tells the compiler that two successive reads or a write followed by a read may return two different values, even though there is no intervening synchronization point.

Unlike the pragma SHARED, the pragma VOLATILE does not guarantee indivisible access. To ensure indivisible access for a variable in your program, you must ensure that sharing of the variable is synchronized by tasking events or a write to a variable for which pragma SHARED has been specified.

The following example explains the difference between shared and volatile variables.

Suppose that you have an access variable named PTR, which you use to control a loop that is executed by a task:

```
while PTR /= null loop
   delay 1.0;
end loop;
```

If you did not declare PTR with the pragma VOLATILE or SHARED, another task could write a nonnull value into PTR's location, and the loop would repeat forever. The loop will also repeat forever if it is checking a value of PTR that was read before the loop was entered.

If PTR is declared with the pragma VOLATILE, then the loop will repeat only until a synchronization point is reached by the task that wrote PTR (the writing task may be running on a different processor, and the new value is not guaranteed to be made visible to other processors until the synchronization point). Also, the value read for PTR may not equal null, or the value read may have half the bit pattern of null and half the bit pattern of the new value, in which case, the value may not be a legal access value.

If PTR is declared with the pragma SHARED, then the loop will not repeat indefinitely even though the task that wrote PTR did not reach another synchronization point. Also, the update and read are guaranteed to be indivisible.

You can use the pragmas VOLATILE and SHARED together to coordinate the sharing of information among tasks. For example:

```
INFO : INFORMATION_RECORD;
pragma VOLATILE(INFO);

INFO_VALID : BOOLEAN := FALSE;
pragma SHARED(INFO_VALID);
 . . .

INFO.SOME_FIELD := SOME_VALUE;
INFO_VALID := TRUE;
```

In this example, the pragma VOLATILE ensures that when INFO.SOME_FIELD is assigned the value SOME_VALUE, the value is stored in the storage area allocated for INFO.SOME_FIELD, and not into a temporary copy or a register. The pragma SHARED makes the assignment to INFO_VALID a synchronization point, thus guaranteeing that the values for INFO and INFO_VALID will both be visible to other tasks.

The pragma SHARED makes it possible for a task to poll the value of INFO_VALID while waiting to access INFO from another task. However, because polling is a kind of busy waiting that takes a fair amount of CPU time, it is usually much better to use a synchronized event to determine completion. For example, you can synchronize a task with event flag wait completion, AST delivery, rendezvous with another task, and so on.

8.4.8 Reentrancy

In most VAX languages, three kinds of reentrancy are possible: serial, recursive, and AST reentrancy. A fourth kind, full reentrancy, is important for Ada programs that have tasks.

A routine is *serially reentrant* if it must execute to completion before it is allowed to be called again. FORTRAN routines are usually serially reentrant.

To understand *recursive reentrancy*, consider a routine that executes to the point of another call to itself; it makes the call to itself (a recursive call), and then continues to make recursive calls until a statement is executed or a condition occurs that ends the recursion. Then, the statements after the point of the recursive call execute, until finally the original call completes. If no calls are permitted until the original call has completed, the routine is said to be recursively reentrant. A recursively reentrant routine is also serially reentrant.

AST reentrancy means that at a random point during the execution of a routine, an AST can occur and the routine may be reentered (by the AST call). The VMS operating system does not normally allow more than one AST service routine to be called at a time for any given access mode. So, if a routine is AST-reentrant, it may be designed to permit at most two calls to be in progress at any one time. An AST-reentrant routine is also serially reentrant.

A routine is *fully reentrant* if it gives correct results when called by multiple tasks whose execution can be suspended at arbitrary points (and resumed in arbitrary orders) in the routine's code. A routine that is fully reentrant is also necessarily AST reentrant, recursively reentrant, and serially reentrant.

The following sections discuss reentrancy in the context of mixed-language programs involving Ada tasks. Unless explicitly qualified, the term *reentrant* denotes *full reentrancy*.

8.4.8.1 Reentrancy in Mixed-Language Tasking Programs

You must be careful when calling non-Ada routines from VAX Ada tasks, because the results will be unpredictable if the routines are not fully reentrant. All VMS system service and most VMS Run-Time Library routines are fully reentrant. In particular, most language-independent Run-Time Library routines (LIB$, MTH$, OTS$, and STR$ routines) are fully reentrant (see the *VMS Run-Time Library Routines Volume*). However, any routine that modifies variables outside its immediate scope or that modifies variables

allocated in static storage is potentially nonreentrant. Also, any language-dependent run-time library routines may be nonreentrant. For example, the FORTRAN run-time library is only AST (not fully) reentrant.

8.4.8.2 Avoiding Nonreentrancy

The subprogram in Example 8–8 shows that if you allow a nonreentrant Ada subprogram (or non-Ada routine) to be reentered, the results can be unpredictable.

Example 8–8: A Nonreentrant Subprogram

```
package CONTAINER is

   -- Function to return 1 + I (its argument).
   --
   function NONREENTRANT(I : INTEGER) return INTEGER;

end CONTAINER;

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

package body CONTAINER is

   GLOBAL_VARIABLE : INTEGER := 0;
   function NONREENTRANT(I : INTEGER) return INTEGER is
   begin
      GLOBAL_VARIABLE := I;              -- Statement S1.
      return (GLOBAL_VARIABLE + 1);      -- Statement S2.
   end NONREENTRANT;

begin
   null;
end CONTAINER;
```

In Example 8–8, the function NONREENTRANT returns 1 plus the value of its argument, I. NONREENTRANT is a serially reentrant subprogram. However, it cannot be called simultaneously by multiple tasks and still produce correct results. For example, consider the following sequence of events:

- The subprogram NONREENTRANT is called by task A, which passes a value of 3 for I.

- A is interrupted just before statement S2 because a higher priority task, B, has become ready.
- Then, B calls NONREENTRANT and passes a value of 1000 for I (that is, B reenters the subprogram while a previous call is in progress).

Although the execution of NONREENTRANT by A sets GLOBAL_VARIABLE to 3, the intervening execution by B changes the global variable to 1000. When task A finally resumes execution, NONREENTRANT returns a value of 1001, instead of the correct answer, which is 4.

There are three ways to avoid the problem shown in Example 8–8:

- Write the routine or subprogram so that it is reentrant.
- Ensure that only one task can call the nonreentrant routine or subprogram.
- Serialize the calls to the nonreentrant routine or subprogram (see Example 8–10 at the end of this section).

To code a reentrant subprogram in VAX Ada, make sure it does not modify any nonlocal or static variables and make sure that it does not call a nonreentrant subprogram (or routine). If you import a non-Ada routine, be aware that it can be reentered if it is imported several times in the same Ada program or if it is imported once and then called from different tasks.

In Example 8–9, the function NONREENTRANT from the Example 8–8 is rewritten so that it is reentrant. Advantage has been taken of the fact that each time a subprogram is entered, its local variables are allocated on the stack. If tasks A and B were to call the following subprogram, each activation of function REENTRANT would create a separate copy of LOCAL_VARIABLE, and interference would not be possible.

Example 8–9: A Reentrant Subprogram

```
function REENTRANT(I : INTEGER) return INTEGER is
   LOCAL_VARIABLE : INTEGER := 0;
begin
   LOCAL_VARIABLE := I;                    -- Statement S1.
   return (LOCAL_VARIABLE + 1);            -- Statement S2.
end REENTRANT;
```

The second solution is to structure your program to ensure that the nonreentrant subprogram can only be called by a single task at a time. For example, if a procedure is defined in the declarative region of the same task that calls it, and the task creates no dependent tasks, then the subprogram cannot be reentered.

The third solution applies especially to existing nonreentrant Ada subprograms, non-Ada routines, or software over which you have no control. You can use a task to prevent reentry by serializing the calls to the reentrant code so that it cannot be reentered. Example 8–10 shows one way to perform serialization.

Example 8–10: Using a Serializing Task to Prevent Reentry

```
package FIX_IT is

   -- This function should be called instead of
   -- NONREENTRANT.  It too returns 1 + I (its argument).
   --
   function ADD_ONE (I : INTEGER) return INTEGER;

end FIX_IT;

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

with CONTAINER;
use CONTAINER;
package body FIX_IT is

   task SERIALIZER is
      entry DO_CALL(I : INTEGER; J : out INTEGER);
   end;

   task body SERIALIZER is
   -- This task calls NONREENTRANT and ensures that it
   -- cannot be reentered.
   --
   begin
      loop
         select
            accept DO_CALL(I : INTEGER; J : out INTEGER) do
               J := NONREENTRANT(I);
            end;
         or
            terminate;
         end select;
      end loop;
   end;
```

(continued on next page)

Example 8–10 (Cont.): Using a Serializing Task to Prevent Reentry

```
   function ADD_ONE (I : INTEGER) return INTEGER is
      RESULT : INTEGER;
   begin
      SERIALIZER.DO_CALL(I, RESULT);
      return RESULT;
   end;

end FIX_IT;
```

In Example 8–10, the task SERIALIZER calls a nonreentrant subprogram in the body of an accept statement. All calls to the nonreentrant code go through the intermediate call to ADD_ONE, and the function NONREENTRANT cannot be reentered.

You can also use a serializing task to allow nonreentrant routines or subprograms to be called from multiple tasks. The serializing task prevents reentry, but you must make sure that it makes all of the calls. This method is recommended when you call any routine or subprogram whose reentrancy is uncertain and you cannot guarantee that reentrant calls will not be attempted.

8.5 Calling VMS System Service Routines from Tasks

VAX Ada provides the package STARLET (see Chapter 6) as well as import-export pragmas (see Chapter 5) to allow you to call VMS system services and make VMS RMS requests directly from an Ada program. In addition, VAX Ada provides a package of selected asynchronous system routines—TASKING_SERVICES—to make such routine calls easier to make from tasks. The following sections discuss the implications of calling system routines from tasks.

If you are coding system services that involve ASTs, see also Section 8.6.

8.5.1 Effects of System Service Calls on Tasks

When you call VMS system services from an Ada program, your process is not totally "blocked." Most system services that put your process in a wait state permit that wait state to be interrupted by ASTs (see the *Introduction to VMS System Services*). To VAX Ada, a task that has entered a VMS wait

state appears to be continuing to execute (because VAX Ada does not in any way intercept system services); VAX Ada does not know that the task is in any way blocked.

Thus, the only tasks that can execute while the system service is executing are tasks that have higher priorities than the calling task, or, if time slicing is in effect, tasks that have a priority equal to the calling task. (The transfer of control to these other tasks can occur when an AST for one of these tasks is delivered to the VAX Ada run-time library—for example, when a delay or time slice expires, an Ada input-output request completes, or an AST is delivered to a task entry specified with the pragma AST_ENTRY.)

This default behavior is not necessarily bad, because waiting for the system service to complete is the default behavior of most nontasking VMS programs. Indeed, if the request is satisfied quickly, allowing any other task to execute could be wasted effort.

You may, however, wish to increase concurrency and allow tasks of lower priority to execute while a higher priority task is in a VMS wait state. Provided that the system service request takes a sufficiently long time, this strategy can allow your program to do more useful work in the same elapsed time.

VAX Ada provides you with two methods for increasing concurrency during a VMS system service wait interval. One method is to have tasks that call time-consuming system services use asynchronous system services or asynchronous VMS RMS services. Then, your program can do other work until it has to handle the resulting VMS ASTs that signify completion of the request. Handling ASTs is a very general and powerful way to increase concurrency, but it also requires more detailed programming. See Section 8.6.

The second method for increasing concurrency is to use the VMS system-routine operations provided in the VAX Ada package TASKING_SERVICES. Like the system routines provided in the package STARLET, the operations in this package provide an interface to a variety of VMS system service and RMS routines. But the operations in the package TASKING_SERVICES are designed to suspend (in the Ada sense) the calling task if the request cannot be immediately satisfied. Other ready tasks (including lower priority tasks) in your program are free to execute or continue executing.

The operations in the package TASKING_SERVICES increase concurrency by calling the asynchronous form of a system service routine (for example, SYS$QIO instead of SYS$QIOW), and then suspending the task and using an AST to signal when the service has completed and the execution of the task can resume. The details of AST handling are hidden by the package.

While this package can help you increase concurrency, in many cases, it has two limitations: you cannot use the operations in the package TASKING_SERVICES to specify an AST routine address or an AST parameter. If your application depends on being able to use such information, you may wish to do your own AST handling as described in Section 8.6.

The specification of the package TASKING_SERVICES is presented in Appendix B.

8.5.2 System Services Requiring Special Care

Certain system services are especially likely to interfere with Ada programs that use tasks, ASTs, or the package TASKING_SERVICES (see Section 8.5.1). You should either avoid or use extra care when using the following services:

SYS$SETAST	STARLET.SETAST
SYS$HIBER	STARLET.HIBER
SYS$EXIT	STARLET.EXI
SYS$DCLEXH	STARLET.DCLEXH

Because they affect a VMS process, these services have a global effect on all tasks of the program. For example, SYS$SETAST prevents delivery of ASTs. Because the VAX Ada run-time library relies heavily on the use of ASTs (they are used to implement delay statements, time slicing, input-output, and so on), disabling ASTs with SYS$SETAST can cause deadlocks (see Example 8–11). This effect can cause these tasks to stall until ASTs are reenabled.

If you must use SYS$SETAST, do not take any of the following actions while ASTs are disabled:

- Execute Ada input-output statements (for example, TEXT_IO.PUT_LINE).
- Execute a delay statement.
- Propagate an unhandled exception.
- Execute any of the tasking operations described in Chapter 9 of the *VAX Ada Language Reference Manual* (for example, make entry calls, execute accept or select statements, and so on). In other words, do not create or wait for dependent tasks.

Example 8–11: Deadlock Caused by a Call to SYS$SETAST

```
procedure SETAST_DEADLOCK is

   task T is
      entry E;
   end;

   task body T is
   begin
      delay 10.0;
      accept E;
   end;

   -- Procedure to set AST enablement to SETTING.
   --
   procedure SETAST(SETTING : BOOLEAN) is separate;

begin

   SETAST(FALSE);
   T.E; -- At this point, task T is delayed, waiting
        -- for the timer AST that signifies the end of
        -- the wait.  The following entry call must suspend
        -- because the task has not reached the accept statement.
        -- But, because the call to SETAST has disabled ASTs,
        -- the delay will never complete, and thus neither
        -- will this entry call.
   SETAST(TRUE);

end SETAST_DEADLOCK;
```

- Busy wait on a flag (a variable) that is to be set by another task.
- Call a subprogram that involves any of the preceding actions.

SYS$HIBER suspends execution of a VMS process. The VAX Ada run-time library uses SYS$HIBER to make your process hibernate when there are no currently ready tasks. If your program also uses SYS$HIBER, make sure that the SYS$WAKE it is waiting for is entered only when the process is waiting for your SYS$HIBER request (and not for the SYS$HIBER request of the VAX Ada run-time library)—the SYS$WAKE you enter may be consumed by the call to SYS$HIBER of VAX Ada, and your hibernating process will not wake up.

SYS$EXIT causes an unconditional program exit. In particular, it does not wait for dependent tasks to terminate normally. Thus, by using SYS$EXIT, you may prevent your Ada program from executing code that it would otherwise execute normally. Unless you are careful that all tasks are terminated or in a state where termination is not needed, the results can be unpredictable. See Example 8–12.

Example 8–12: Unpredictability of SYS$EXIT

```
with TEXT_IO; use TEXT_IO;
procedure PULL_THE_RUG_OUT is

   pragma TIME_SLICE (1.0);

   type CONDITION is new INTEGER;

   STATUS : CONDITION;

   task T;
   task body T is
   begin
      for I in 1..10 loop
         PUT_LINE("I'm T and I'm not done yet.");
         delay 1.0;
      end loop;
      PUT_LINE("T is done now.");
   end;

   procedure DO_EXIT(STATUS      : out CONDITION;
                     EXIT_STATUS : in CONDITION := 1);
   pragma INTERFACE(VMS, DO_EXIT);
   pragma IMPORT_VALUED_PROCEDURE(DO_EXIT,
                                  "SYS$EXIT",
                                  MECHANISM => (VALUE, VALUE));

begin

   delay 5.0;
   PUT_LINE("Pulling the rug out from T NOW.");
   DO_EXIT(STATUS);

end PULL_THE_RUG_OUT;
```

If you use SYS$DCLEXH to establish exit handlers, make sure you understand that not all Ada operations can be executed reliably from VMS exit handlers. Thus, there are some restrictions on exit handlers written in Ada. These restrictions are the same as those for AST handlers, and they stem from the fact that exit handlers can be invoked asynchronously, such as when you press CTRL/Y at the terminal. (See Section 8.6.3 for the restrictions on using AST handlers.)

Specifically, a VMS exit handler written in Ada must not take any of the following actions:

- Execute an Ada input-output statement.
- Execute a delay statement.
- Propagate an unhandled exception.
- Execute any tasking operation.

- Busy wait on a flag (a variable) that is to be set by another task.
- Call a subprogram that involves any of the preceding actions.

Note that the Ada run-time library makes use of a special input-output exit handler that flushes input-output buffers (those that are unlocked) at the time of exit, so user exit handlers should not be needed for the purpose of flushing and closing files. Another good reason for avoiding Ada exit handlers is that they are nonportable.

8.6 Handling Asynchronous System Traps (ASTs)

ASTs are a way for the VMS operating system or VMS RMS to notify a process (which may be actively executing instructions) that some event has occurred. Many VMS system services allow you to specify that an AST be delivered when the service completes or when some event related to the service occurs. Such services often have two forms:

- A synchronous form that forces the process to wait until the service is completed
- An asynchronous form that initiates the service, and immediately returns, allowing the process to continue and the service to be completed independently

For example, the synchronous VMS system service SYS$QIOW makes the process wait until the service completes, but the asynchronous form, SYS$QIO, does not; both forms allow you to specify an AST service routine.

By handling ASTs from your Ada program, you can increase concurrency during the process wait states that result after your program executes certain system service or VMS RMS requests. Before you decide to handle ASTs directly, you should investigate the package TASKING_SERVICES to see if you can use any of its operations instead (the operations in the package TASKING_SERVICES are more convenient to use); see Section 8.5.1.

8.6.1 The Pragma AST_ENTRY and the AST_ENTRY Attribute

You handle ASTs in VAX Ada with the AST_ENTRY pragma and AST_ENTRY attribute.

For a formal description of the AST_ENTRY pragma and attribute, see Chapter 9 of the *VAX Ada Language Reference Manual*. Informally, the AST_ENTRY pragma and attribute provide a mechanism that transforms the delivery of an AST into a special kind of entry call. As in an ordinary entry call, if the task does not immediately accept the call, the AST entry call becomes enqueued on the entry. For example:

```
with STARLET; use STARLET;
 . . .
task HANDLER is

   -- Entry RECEIVE_AST can receive AST entry calls as
   -- well as normal entry calls.
   --
   entry RECEIVE_AST;
   pragma AST_ENTRY(RECEIVE_AST);

end HANDLER;
 . . .
   -- The AST_ENTRY attribute supplies QIO's ASTADR parameter
   -- with the address of a special AST handler that will
   -- schedule an entry call to RECEIVE_AST.
   --
   QIO( . . .
       ASTADR => HANDLER.RECEIVE_AST'AST_ENTRY,
        . . . );
 . . .
```

An AST entry call acts as if the call were made by a task that has a priority of 8. In accordance with Ada rendezvous rules, the statement list of the accept statement for this entry is executed with the higher of this priority and the priority of the accepting task.

The AST parameter passed to the system service (for example, the astprm argument of the SYS$QIO system service) and later delivered by the AST is, in turn, passed to the accept statement (if a formal parameter is specified). For example:

```
with STARLET; use STARLET;
 . . .
task HANDLER is

   -- Entry RECEIVE_AST expects to receive an
   -- AST parameter.
   --
   entry RECEIVE_AST(X : INTEGER);
   pragma AST_ENTRY(RECEIVE_AST);

end HANDLER;
```

```
task body HANDLER is
   . . .
begin
   accept RECEIVE_AST(X:INTEGER) do
    . . .
   end RECEIVE_AST;
end HANDLER;
 . . .
   -- QIO's ASTPRM parameter (with a value of 33) is
   -- passed to the accept statement as parameter X
   -- when the rendezvous occurs. (An AST entry can
   -- receive only zero or one parameter.)
   --
   QIO( . . .
        ASTADR => HANDLER.RECEIVE_AST'AST_ENTRY,
        ASTPRM => 33);
 . . .
```

Thus, to handle ASTs, you must first specify which entry in a task type can receive AST entry calls. You do this by specifying the entry with the pragma AST_ENTRY when you declare the task type. Only single entries (not entry families) can receive AST entry calls and, therefore, only single entries can be named in the pragma AST_ENTRY.

To specify an AST service routine, you must use the AST_ENTRY attribute. The AST_ENTRY attribute takes a task name and entry as parameters and returns an address of a special service routine created by the VAX Ada run-time library. Then, when the AST occurs, the special service routine is called; the routine enqueues the AST parameter in a special way on the requested entry, making the enqueuing look like an entry call, and then the routine returns from the AST call.

Once the AST parameter is enqueued on the entry, the rendezvous can occur. The rendezvous is subject to the Ada rendezvous rules (see Chapter 9 of the *VAX Ada Language Reference Manual*), and so may not occur immediately. Also, the rendezvous is performed after the special service routine has returned. Thus, other ASTs are not inhibited. (This behavior is required by the nature of ASTs and the nature of Ada rendezvous.)

If the entry call were made directly from the AST service routine, no other ASTs could be delivered until the rendezvous had completed, and unpredictable deadlocks could result. Such deadlocks still could develop if a non-Ada program were to call an Ada program from an AST service routine. See Section 8.6.3 for more information.

8.6.2 Constraints on Handling ASTs

Any AST delivered to a task that is completed or abnormal is ignored. In other words, the AST is ignored if it occurs for some entry of a task that is not callable but is not yet terminated (both T'TERMINATED and T'CALLABLE are FALSE).

If an AST occurs for an entry of a task that is terminated (T'TERMINATED is TRUE), then the program is erroneous and execution is unpredictable. The VAX Ada run-time library may not detect this situation; you must code your application so that an AST cannot occur for an entry in a terminated task.

Each time an AST is delivered, the VAX Ada run-time library allocates a block of storage (an AST packet) to hold the AST parameter, and the storage is enqueued on the entry to which the AST applies. This block of storage is released only after the rendezvous has completed. If your program generates ASTs at a higher rate than it accepts AST entry calls, the total amount of storage allocated can become very high. To reduce the amount of storage consumed, write any AST-handling programs so that they accept an AST for every AST generated. You can do this easily by having the same task that accepts the AST entry call also generate the next AST. In this manner, you can limit the amount of storage consumed by pending AST entry calls.

Another way to prevent this problem is to extend the size of the AST packet pool available to your program, using the package SYSTEM_RUNTIME_TUNING. See Appendix B for more information on this package and its operations.

8.6.3 Calling Ada Subprograms from Non-Ada AST Service Routines

Be very careful when using an AST service routine, or when calling an Ada subprogram from an AST service routine. If the Ada subprogram performs certain kinds of Ada operations, including input-output operations or task-related operations, a deadlock can develop (VAX Ada itself uses ASTs to perform these operations). If you call such an Ada subprogram from an AST service routine, or use it as an AST service routine, your program can develop a deadlock with the characteristics that one or more tasks are suspended indefinitely and ASTs can no longer be delivered.

For example, consider the following situation:

- You call Ada subprogram P from a non-Ada AST service routine; the AST may be delivered at any time.
- When the AST is delivered, the main task or a task Q in your program may have already allocated a resource that will be needed by P. In addition, Q could be currently suspended, awaiting the delivery of an AST.
- Because the resource is not available to P, the VAX Ada run-time library has no choice but to suspend the execution of P and switch control to another ready task.
- The invocation of P occurred when the ASTs were disabled, so they remain disabled after P is suspended.

A deadlock has developed in this situation for the following reasons:

- P cannot proceed until the resource becomes available.
- The resource cannot be released by Q until ASTs are delivered.
- ASTs cannot be delivered until P and its caller return control back to the VMS environment.

All of this occurs because the VMS operating system *implicitly* disables all AST delivery while an AST-handling routine is active.

Thus, you should handle ASTs in VAX Ada as described in Section 8.6.1. If you must write AST routines in Ada, then obey the following rules to avoid the kind of deadlock described in this section. Your routine must not take any of the following actions:

- Execute an Ada input-output statement (for example, TEXT_IO.PUT_LINE). In other words, your routine must not use any of the input-output operations defined in Chapter 14 of the *VAX Ada Language Reference Manual* or in the VAX Ada package TASKING_SERVICES.
- Execute a delay statement.
- Propagate an unhandled exception.
- Execute any of the tasking operations described in Chapter 9 of the *VAX Ada Language Reference Manual* (for example, make entry calls, execute accept or select statements, and so on). In other words, you must not create or wait for dependent tasks.
- Busy wait on a flag (a variable) that is to be set by another task.
- Call a subprogram that involves any of the preceding actions.

8.6.4 Examples of Handling ASTs from Ada Programs

Examples 8–13 and 8–14 show the use of the pragma AST_ENTRY and the AST_ENTRY attribute.

Example 8–13: Simple Use of the Pragma AST_ENTRY and the AST_ENTRY Attribute

```
with TEXT_IO, SYSTEM, CONDITION_HANDLING, STARLET;
procedure TRY_ASTS is

   STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;

   package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);

   -- Task that will handle the ASTs activated by the main program.
   --
   task AST_HANDLER is
      entry RECEIVE_AST(X : INTEGER);
      pragma AST_ENTRY(RECEIVE_AST);
   end AST_HANDLER;

   task body AST_HANDLER is

      FORE : constant TEXT_IO.FIELD := 3;

   begin
      loop
         select
            accept RECEIVE_AST(X : INTEGER) do
               INT_IO.PUT(X, FORE);
                end RECEIVE_AST;
            or
               terminate;
            end select;
      end loop;
   end AST_HANDLER;

begin
```

(continued on next page)

Example 8–13 (Cont.): Simple Use of the Pragma AST_ENTRY and the AST_ENTRY Attribute

```
   -- Queue 20 ASTs to be activated, and give each an index.
   --
   for I in 1..20 loop
      STARLET.DCLAST(
         STATUS,
            -- Condition value returned.
         AST_HANDLER.RECEIVE_AST'AST_ENTRY,
            -- Entry to receive the AST.
         STARLET.USER_ARG_TYPE (I));
            -- The AST parameter.

      -- If DCLAST fails to queue an AST, then raise the error.
      --
      if not CONDITION_HANDLING.SUCCESS(STATUS) then
         CONDITION_HANDLING.STOP (STATUS);
      end if;
   end loop;

end TRY_ASTS;
```

Example 8–14: Using an AST Entry to Intercept a CTRL/C

```
-- Package specification for CONTROL_C_HANDLING.
--
package CONTROL_C_HANDLING is
end CONTROL_C_HANDLING;

-- Package body for CONTROL_C_HANDLING.
--
with SYSTEM, TEXT_IO, CONDITION_HANDLING,
     STARLET, UNCHECKED_CONVERSION;
package body CONTROL_C_HANDLING is

  -- Used to specify an outband character to QIOW.
  --
  type SHORT_FORM_TERMINATOR is
     record
        ZERO : SYSTEM.UNSIGNED_LONGWORD;
        MASK : SYSTEM.UNSIGNED_LONGWORD;
     end record;
```

(continued on next page)

Example 8–14 (Cont.): Using an AST Entry to Intercept a CTRL/C

```
   CONTROL_C       : constant SYSTEM.UNSIGNED_LONGWORD
      := SYSTEM.UNSIGNED_LONGWORD(2**CHARACTER'POS(ASCII.ETX));

   TERMINATOR_MASK : constant SHORT_FORM_TERMINATOR
      := (ZERO => 0, MASK => CONTROL_C);

   STATUS          : CONDITION_HANDLING.COND_VALUE_TYPE;
   STATUS1         : CONDITION_HANDLING.COND_VALUE_TYPE;
   CHAN            : STARLET.CHANNEL_TYPE;
   TERM_DEV        : constant STARLET.DEVICE_NAME_TYPE := "TT:";

   -- This task services CONTROL_C outband ASTs.
   --
   task AST_SERVER is
      entry CONTROL_C_HANDLER;
      pragma AST_ENTRY(CONTROL_C_HANDLER);
   end AST_SERVER;

   task body AST_SERVER is
   begin
      loop
         select
            accept CONTROL_C_HANDLER do
               TEXT_IO.PUT_LINE("Control_C was received.");
            end CONTROL_C_HANDLER;
         or
            terminate;
         end select;
      end loop;
   end AST_SERVER;

   function FROM_AH_TO_UL is
      new UNCHECKED_CONVERSION (SYSTEM.AST_HANDLER,
                                SYSTEM.UNSIGNED_LONGWORD);

begin

   -- Assign a channel to the terminal.
   --
   STARLET.ASSIGN(STATUS,    -- Condition value returned.
                  TERM_DEV, -- Terminal device to assign to.
                  CHAN);    -- Channel number.

   if not CONDITION_HANDLING.SUCCESS(STATUS) then
      CONDITION_HANDLING.STOP(STATUS);
   end if;
```

(continued on next page)

Example 8–14 (Cont.): Using an AST Entry to Intercept a CTRL/C

```
   -- Enable outband ASTs for CONTROL_C; direct the ASTs
   -- to AST_SERVER.
   --
   STARLET.QIOW(
      STATUS => STATUS,
      CHAN => CHAN,
      FUNC => SYSTEM."OR"(STARLET.IO_SETMODE, STARLET.IO_M_OUTBAND),
      P1   => FROM_AH_TO_UL(AST_SERVER.CONTROL_C_HANDLER'AST_ENTRY),
      P2   => SYSTEM.TO_UNSIGNED_LONGWORD(TERMINATOR_MASK'ADDRESS));

   if not CONDITION_HANDLING.SUCCESS(STATUS) then
      STARLET.DASSGN(STATUS => STATUS1, CHAN => CHAN);
      CONDITION_HANDLING.STOP(STATUS);
   end if;

end CONTROL_C_HANDLING;

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

-- A program that uses the package CONTROL_C_HANDLING.
--
with CONTROL_C_HANDLING;
with TEXT_IO; use TEXT_IO;
procedure TRY_CONTROL_C is
begin

   PUT_LINE("Press any number of CTRL/Cs for " &
            "the next 30 seconds.");
   PUT_LINE("CTRL/Cs are trapped and " &
            "serviced by CONTROL_C_HANDLING.");
   delay 30.0;
   NEW_LINE;
   PUT_LINE("Main program terminating  . . .  ");

end TRY_CONTROL_C;
```

8.7 Measuring and Tuning Tasking Performance

When you use tasks in your program, you must frequently trade off between responsiveness and throughput. Responsiveness is how fast a task responds to an asynchronous event, such as a user typing at a keyboard. Throughput is how much useful work, as measured by CPU time, a program accomplishes in a given amount of elapsed time (time spent switching tasks is overhead and takes CPU cycles that could be used for useful work).

In general, if you enable time slicing with the pragma TIME_SLICE, you are increasing responsiveness at the expense of more task-switching overhead and therefore decreased throughput. Smaller values of the time-slice interval represent higher amounts of this overhead.

Similarly, if you assign a higher priority to a task, you are opting for responsiveness rather than throughput. Assigning a higher priority to some task invariably means that the program will perform more task switches—every time the high priority task becomes eligible for execution, Ada rules require that it displace a currently running lower priority task.

In a large program that has many tasks, not all of the effects of changing the program are immediately obvious. To help you measure the effects of a change, VAX Ada provides the VMS Debugger commands SHOW TASK/STATISTICS and SHOW TASK/FULL. See *Developing Ada Programs on VMS Systems* for information on debugging Ada tasks.

Chapter 9

Improving Run-Time Performance

To write VAX Ada programs that compile and execute efficiently, you should be aware of certain compiler and language features that can affect code size, as well as program compilation and execution times. This chapter discusses the following topics:

- Compiler optimizations
- Inline expansion of subprograms
- Improving the performance of generics
- Techniques for reducing CPU time and elapsed time

9.1 Compiler Optimizations

The VAX Ada compiler performs a number of standard optimizations to improve the quality of the generated code. For example, the compiler performs the following optimizations:

- Elimination of some common subexpressions
- Code hoisting from structured statements, including the removal of invariant computations from loops
- Inline code expansion for many predefined operations
- Rearranging of unary minus and **not** operations to eliminate unary negation/complement operations
- Partial evaluation of logical expressions
- Global assignment of variables to registers

- Forward propagation of constant values
- Reordering of the evaluation of expressions to minimize the number of temporary values required
- Peephole optimization of instruction sequences

In addition, the compiler performs the following Ada-specific optimizations:

- Elimination of redundant constraint checks
- Evaluation of all static subexpressions, even when evaluation is not required by the language; also evaluation of other compile-time constant expressions that may not be considered to be "static" expressions in the language (for example, expressions involving catenation or attributes such as T'IMAGE)
- Elimination of dead code (for example, elimination of unreachable branches with compile-time constant selectors in if and case statements)
- Elimination of redundant bounds checking of arrays in array subscripting and slicing
- Elimination of redundant address evaluations

In cases where address evaluations are eliminated, the address is evaluated only once and a special increment instruction is generated. The address of A(I) in the following example is such a case:

```
A(I) := A(I) + 1;
```

Note that the result of this addition may be assigned to a temporary variable if a constraint check must be performed before A(I) is updated.

In addition, in matrix computations like the following (A(I,J)), the address calculation tends to be calculated each time using special VAX hardware addressing for this purpose (context indexing):

```
for I in 1 .. N loop
   for J in 1 .. M loop
      A(I,J) := SOME_VALUE;
   end loop;
end loop;
```

Depending on various details, strength reduction may alternately be used.

9.2 Using the Pragma INLINE

To allow you to expand subprograms inline and thereby decrease the amount of time spent in making subprogram calls, the Ada language provides the pragma INLINE. In VAX Ada, the pragma INLINE can affect your program in one of two ways:

- Explicitly—you declare a subprogram to be expanded inline.
- Implicitly—the compiler automatically expands subprogram bodies inline under certain conditions.

Section 9.2.1 gives the conditions for the explicit use of this pragma; Section 9.2.2 gives the conditions under which implicit inline expansion takes place.

See Section 9.2.3 for examples showing the use of the pragma INLINE for a variety of interesting cases.

The decisions made by the compiler for the pragma INLINE are shown in compilation notes messages. In particular, they are shown at calls to the affected subprograms. For example:

```
 . . .
    1   with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
    2   procedure SHOW_INLINE is
    3
    4      type T is new INTEGER range 1..10;
    5      function "+" (X,Y: INTEGER) return INTEGER
    6         renames STANDARD."+";
    7      VAR1,VAR2: T := 3;
    8
    9      function "+" (X,Y: T) return INTEGER is
...........1
%I, (1) Code generation suppressed for function +, which is always
        expanded inline [LRM 6.3.2; RTR 9.2.1]

   10      begin
   11         return INTEGER(X*Y);
   12      end;
   13      pragma INLINE("+");
   14
   15   begin
   16      PUT(VAR1+VAR2);
.............1....2
%I, (1) Call of procedure PUT in INTEGER_TEXT_IO at line 23 (from
        TEXT_IO at line 148) expanded inline [LRM 6.3.2; RTR 9.2]
%I, (2) Call of function + at line 9 expanded inline
        [LRM 6.3.2; RTR 9.2]
```

```
   17   end;
. . .
```

To obtain compilation notes messages, use the /WARNINGS=COMPILATION_NOTES qualifier with the DCL ADA or ACS COMPILE or RECOMPILE commands. See *Developing Ada Programs on VMS Systems* for more information on these commands and this qualifier.

Chapter 6 of the *VAX Ada Language Reference Manual* gives the syntax and placement rules for this pragma.

9.2.1 Explicit Use

A subprogram for which the pragma INLINE has an effect is considered to be *inlinable*. You can use the pragma INLINE to explicitly expand a *subprogram declaration* or *body* inline only under the following conditions.

- The parameters can be of any type except the following:
  - A task type
  - A composite type that has components of a task type

  Function results can be of any type except the following:
  - A task type
  - A composite type that has components of a task type
  - An unconstrained array type
  - An unconstrained type with discriminants (with or without defaults)
- The body of the subprogram cannot contain any of the following:
  - A subprogram body, task or generic declaration or body stub (a subprogram declaration for an imported subprogram is allowed)
  - A package body (a package specification is allowed)
  - A generic instantiation
  - An exception declaration
  - An access type declaration (a type derived from an access type is allowed)
  - An array or record type declaration
  - Any dependent tasks (that is, any constant or variable declaration that implies the creation of a task)

- Any subprogram call that denotes the given subprogram (direct recursion) or any containing subprogram, either directly or by means of a renaming

You can use the pragma INLINE for an *implicit operator declaration* or for a *derived subprogram declaration*, as follows:

- If you use the pragma INLINE for an implicit operator declaration, the pragma is accepted but has no effect. "Calls" of implicit operators are implemented by inline code in nearly all cases. (Where VMS Run-Time Library routines are used, as for some exponentiations, no alternative inline code sequence is provided.)
- If you use the pragma INLINE for a derived subprogram declaration, the pragma is accepted but has no effect. Calls of derived subprograms are implemented as inline type conversions preceding and/or following a call of the parent subprogram as appropriate to the formal parameters and, in the case of a function, the result. The call of the parent subprogram is expanded inline according to whether a pragma INLINE has been given (explicitly or implicitly) for that parent subprogram and whether the parent subprogram itself is inlinable.

You can use the pragma INLINE for a *generic subprogram instantiation*, for a *generic subprogram declaration*, or for a *subprogram body stub declaration*, as follows:

- If you use the pragma INLINE for a generic subprogram instantiation, the resulting subprogram must satisfy the preceding restrictions. You can use the pragma INLINE for an instantiation of a predefined generic declaration (such as for UNCHECKED_CONVERSION), but you will not achieve any benefit because such instantiations always result in (implicit) inline code.
- If you use the pragma INLINE for a generic subprogram declaration, the resulting effect is that an implicit pragma INLINE (see Section 9.2.2) then applies to every generic subprogram instantiation of that declaration; that implicit pragma is accepted provided the resulting subprogram satisfies the preceding restrictions. (That is, some instantiations may be inlinable, while others may not be, depending on the characteristics of the generic actual parameters.)
- If you use the pragma INLINE for a subprogram body stub declaration, the subprogram signature must satisfy the preceding restrictions for the parameters and result. Calls of such stubs are never expanded inline within that same unit because the dependent stub is not necessarily available.

If a pragma INLINE applies to a subprogram resulting from an instantiation, and if the instantiation and call are in the same unit, the compiler attempts to expand the instantiation inline so as to expand the subprogram call inline. If the inline expansion of the instantiation is successful, a dependence is established on the generic body. (Do not confuse the inline expansion of instantiations with the inline expansion of subprogram calls; see Section 9.3.1 for more information.)

Note that a pragma INLINE contained within a generic declaration or template is not checked as such. The check occurs, according to the preceding rules, for each instantiation that results in a (nongeneric) subprogram.

Also note that code is usually still generated for an inlinable subprogram to allow for normal calls (possibly in previously compiled units) that cannot be or were not expanded inline (see the following paragraph). However, if the subprogram qualifies to be implicitly expanded inline (as described in Section 9.2.2), then code is not generated.

A *call to a subprogram* is expanded inline provided that the following are true:

- The subprogram has the pragma INLINE specified and is inlinable.
- The call is not contained in the result of expanding a call of that same subprogram (indirect recursion).
- The subprogram body is available in either the current unit or in the compilation library (the library secondary unit must not be obsolete). (Note that the inline expansion of a subprogram body from a unit in the compilation library creates a dependence on that unit.)

9.2.2 Implicit Use

The VAX Ada compiler may assume an implicit pragma INLINE for a subprogram body that has one or more of the following characteristics:

- Satisfies all of the requirements for an inlinable subprogram (see Section 9.2.1).
- Is local to the current compilation (so that it cannot be called from any other unit). A pragma INLINE may be assumed for subprograms with nonlocal calls, depending on the value of the /OPTIMIZE=INLINE compilation qualifier (see *Developing Ada Programs on VMS Systems*).
- Contains no calls to any other inlinable subprogram.

- Has an estimated code size when expanded inline that is no greater (or only slightly greater) than the call it replaces. (The estimation of size is based on heuristics and is not exact; however, it is designed to give a close approximation.)

When local implicit inline expansion is done, no code is generated for the subprogram declaration and every call is expanded inline. See *Developing Ada Programs on VMS Systems* for more information on how inline expansion affects unit dependences, obsolete units, and recompilation.

9.2.3 Pragma INLINE Examples

The following sections show some special cases of the use of the pragma INLINE and give examples of using it with generics. In particular, note that the placement of the pragma is important with nongeneric subprograms: if the pragma appears after a subprogram specification it has a different effect than when it appears after a subprogram body.

9.2.3.1 Inline Expansion of Subprogram Specifications and Bodies

When you apply the pragma INLINE to an inlinable subprogram specification, inline expansion takes place for any call of the subprogram. For example:

```
package INLINE_SPEC is
   PKG_VAR: INTEGER := 20;
   function INLINED_F (X: INTEGER) return INTEGER;
   pragma INLINE(INLINED_F);
end INLINE_SPEC;

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

package body INLINE_SPEC is
   function INLINED_F (X: INTEGER) return INTEGER is
   begin
      return X*10;
   end;
begin
   PKG_VAR := INLINED_F(PKG_VAR);  -- Expanded inline.
end INLINE_SPEC;

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

```
with INLINE_SPEC; use INLINE_SPEC;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure USE_INLINE_SPEC is
   VAR: INTEGER := 10;
begin
   PUT(INLINED_F(VAR));              -- Expanded inline as long
                                     -- as the package body for
                                     -- the package INLINE_SPEC
                                     -- is available.
   PUT(INLINE_SPEC.PKG_VAR);
end USE_INLINE_SPEC;
```

Here, INLINED_F is expanded inline both in the body of the package INLINE_SPEC and in the procedure USE_INLINE_SPEC, which also calls INLINED_F.

Because a **with** clause makes the specification (not the body) of a subprogram available to another compilation unit, the application of the pragma INLINE to the body of a subprogram causes inline expansion to take place only where the body is visible. Thus, if the package INLINE_SPEC were rewritten so that the pragma INLINE applied to the body of INLINED_F, inline expansion would occur only in the call to INLINED_F in the body of the package in which it was declared:

```
package INLINE_BODY is
   PKG_VAR: INTEGER := 20;
   function INLINED_F (X: INTEGER) return INTEGER;
end INLINE_BODY;

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

package body INLINE_BODY is
   function INLINED_F (X: INTEGER) return INTEGER is
   begin
      return X*10;
   end;
   pragma INLINE(INLINED_F);
begin
   PKG_VAR := INLINED_F(PKG_VAR);    -- Expanded inline.
end INLINE_BODY;

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

with INLINE_BODY; use INLINE_BODY;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure USE_INLINE_BODY is
   VAR: INTEGER := 10;
begin
   PUT(INLINED_F(VAR));              -- Not expanded inline.
   PUT(INLINE_BODY.PKG_VAR);
end USE_INLINE_BODY;
```

When you apply the pragma INLINE to a library subprogram body that does not have a corresponding specification, the effect is the same as the effect you get when you apply the pragma INLINE to a specification. For example:

```
function INLINED_F (X: INTEGER) return INTEGER is
begin
   return X*10;
end INLINED_F;
pragma INLINE(INLINED_F);

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

with INLINED_F;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
procedure USE_INLINED_F is
   VAR: INTEGER := 10;
begin
   PUT(INLINED_F(VAR));    -- Expanded inline.
end USE_INLINED_F;
```

Here, the procedure INLINED_F is expanded inline in the call from the procedure USE_INLINED_F.

9.2.3.2 Inline Expansion of Generic Subprograms

When you apply the pragma INLINE to a generic subprogram, any subsequent instantiations are potentially inlinable (assuming they meet the requirements outlined in Section 9.2.1). For example:

```
generic
   type T is limited private;
procedure GEN_PROCEDURE;
pragma INLINE(GEN_PROCEDURE);

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

procedure GEN_PROCEDURE is
   O: T;
begin
   null;
end GEN_PROCEDURE;

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

with GEN_PROCEDURE;
procedure USE_GEN_PROCEDURE is

   task type TASK_TYPE is end;
   type ARR is array (1..10) of TASK_TYPE;

   procedure INT_PROCEDURE is
      new GEN_PROCEDURE(INTEGER);  -- Inlinable.
   procedure ARR_PROCEDURE is
      new GEN_PROCEDURE(ARR);      -- Not inlinable.
```

```
   task body TASK_TYPE is
   begin
     null;
   end;

begin
   INT_PROCEDURE;  -- Expanded inline.
   ARR_PROCEDURE;  -- Not expanded inline.
end USE_GEN_PROCEDURE;
```

Here, the procedure USE_GEN_PROCEDURE.INT_PROCEDURE is inlinable, and, in fact, is expanded inline when it is called from USE_GEN_PROCEDURE. USE_GEN_PROCEDURE.ARR_PROCEDURE is not inlinable, because it instantiates GEN_PROCEDURE with an array of tasks (anything involving dependent tasks cannot be expanded inline; see Section 9.2.1).

Note that to expand the call to USE_GEN_PROCEDURE.INT_PROCEDURE inline, the procedure USE_GEN_PROCEDURE establishes a dependence on the generic procedure body GEN_PROCEDURE. Because of the dependence, the instantiations USE_GEN_PROCEDURE.INT_PROCEDURE and USE_GEN_PROCEDURE.ARR_PROCEDURE are expanded inline (do not confuse the inline expansion of instantiations with the inline expansion of subprogram calls; for example, see Section 9.3.1). The dependence means that if the body for the generic procedure GEN_PROCEDURE is later compiled again or replaced, the procedure USE_GEN_PROCEDURE will become obsolete and will need to be recompiled. See *Developing Ada Programs on VMS Systems* for more information.

When you apply the pragma INLINE to subprograms that are declared inside a generic package or subprogram, they are potentially inlinable in an instantiation (again, assuming that they meet the requirements outlined in Section 9.2.1). For example:

```
generic
   type T is limited private;
package GEN_INLINE is
   procedure DECLARE_VAR;
   pragma INLINE(DECLARE_VAR);
end GEN_INLINE;

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

package body GEN_INLINE is
   procedure DECLARE_VAR is
      X: T;
   begin
      null;
   end;
end GEN_INLINE;
```

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

with GEN_INLINE;
procedure USE_GEN_INLINE is

   task type TASK_TYPE is end;
   type ARR is array (1..10) of TASK_TYPE;

   package INLINE_INT is new GEN_INLINE(INTEGER); -- Inlinable.
   package INLINE_ARR is new GEN_INLINE(ARR);     -- Not inlinable.

   task body TASK_TYPE is
   begin
      null;
   end;

begin
   INLINE_INT.DECLARE_VAR;  -- Expanded inline.
   INLINE_ARR.DECLARE_VAR;  -- Not expanded inline.
end USE_GEN_INLINE;
```

Here, procedure DECLARE_VAR is inlinable in the instantiation INLINE_INT; it is not inlinable in the instantiation INLINE_ARR (again, because INLINE_ARR involves tasks). Thus, the call to INLINE_INT.DECLARE_VAR expands DECLARE_VAR inline; the call to INLINE_ARR.DECLARE_VAR does not.

Note that to expand the call to INLINE_INT.DECLAR_VAR inline, the procedure USE_GEN_INLINE establishes a dependence on the generic package body GEN_INLINE. Because of the dependence, the instantiations INLINE_INT.DECLARE_VAR and INLINE_ARR.DECLARE_VAR are expanded inline.

9.3 Making Use of Generics

VAX Ada offers a number of features that allow you to improve the compilation time and performance of programs that use generics. For example:

- You can control how code is generated for generics by using the pragmas INLINE_GENERIC and SHARE_GENERIC or by using a number of equivalent /OPTIMIZE compilation qualifier options.

  The pragma INLINE_GENERIC causes the compiler to expand the generic body inline at the point of instantiation. The pragma SHARE_GENERIC causes the compiler to generate code that can be shared by several instances of the same generic. Table 9–1 compares the effects of these two pragmas with the default behavior.

The decisions made by the compiler for these pragmas are shown in compilation notes messages, which you can obtain with the /WARNINGS=COMPILATION_NOTES qualifier at compile time. See *Developing Ada Programs on VMS Systems* for more information on the /WARNINGS and /OPTIMIZE qualifiers and their options.

- You can use the predefined library-level instantiations provided for commonly used generics; for example, LONG_FLOAT_TEXT_IO, LONG_FLOAT_MATH_LIB, and so on.

Table 9–1: Comparison of the Effects of the Pragmas INLINE_GENERIC and SHARE_GENERIC

Effect	Neither Pragma Applies (Default)	Pragma SHARE_GENERIC Applies	Pragma INLINE_GENERIC Applies
Instances are compiled separately from the unit in which the instantiation occurred.	Yes	Yes	No; generic is expanded inline at the point of instantiation
The unit containing the instantiation depends on the unit containing the generic body.	No	No	Yes
The code generated for the instance can potentially be shared by subsequent instances.	No	Usually	No
The instance shares code from previous instances to which the pragma SHARE_GENERIC applied.	Yes, if suitable	Yes, if suitable	No

9.3.1 Using the Pragma INLINE_GENERIC

VAX Ada implements generics so that the bodies resulting from each instance of a generic are compiled separately from the unit in which the generic instantiation occurs. This implementation is similar to the way in which a subunit is compiled separately from its parent unit. It means that a compilation unit that contains an instantiation does not depend on the instantiation's corresponding generic body, and thus does not need to be recompiled when the generic body changes.

You can modify this behavior by specifying a pragma INLINE_GENERIC for the generic declaration or for a particular instance of a generic declaration. Chapter 12 of the *VAX Ada Language Reference Manual* gives the syntax and rules for using the pragma INLINE_GENERIC. For convenience, the syntax is summarized here:

```
pragma INLINE_GENERIC (name {,name});
```

The pragma INLINE_GENERIC causes the compiler to expand the generic body in the unit containing the instantiation provided that the corresponding body has been compiled and is current. Like subprogram inline expansion, generic inline expansion generally optimizes execution time.

Generic inline expansion also changes the dependences among instantiations and generic bodies. For example, a unit containing an instantiation for which the pragma INLINE_GENERIC is in effect may depend on the unit that contains the generic body. The dependence means that if the unit containing the generic body is compiled again or replaced, the unit containing the instantiation becomes obsolete and must be recompiled. See *Developing Ada Programs on VMS Systems* for more information on dependences, obsolete units, and recompilation.

For example:

```
generic
   type ITEM is private;
procedure USE_ITEM (A, B: ITEM);

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

procedure USE_ITEM (A, B: ITEM) is
begin
   null;
end USE_ITEM;

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

with USE_ITEM;
procedure USE_GENERIC_INLINE is

   procedure USE_INTEGER is new USE_ITEM(INTEGER);
   pragma INLINE_GENERIC(USE_INTEGER);

   X,Y: INTEGER;

begin

   USE_INTEGER(X,Y);

end USE_GENERIC_INLINE;
```

In this example, the procedure USE_GENERIC_INLINE depends on the body for the generic procedure USE_ITEM because a pragma INLINE_GENERIC is specified for USE_INTEGER (which is an instantiation of USE_ITEM), and because the generic procedure body USE_ITEM is available (see the restrictions at the end of this section).

You can maximize generic inline expansion either by specifying a pragma INLINE_GENERIC for all instantiations, or by using the /OPTIMIZE=INLINE:GENERICS or /OPTIMIZE=INLINE:MAXIMAL qualifiers at compile time. See *Developing Ada Programs on VMS Systems* for more information on the /OPTIMIZE qualifier and its options.

Maximal generic inline expansion is often most effective in combination with maximal subprogram inline expansion. However, maximal generic inline expansion is usually most effective for applications that contain relatively few generic instantiations. If your application uses generics extensively, you may find that maximal generic inline expansion substantially increases program size. In such cases, generic inline expansion may increase elapsed time at run time because of increased paging.

9.3.2 Using the Pragma SHARE_GENERIC

When generic inline expansion is not in effect, you can use the pragma SHARE_GENERIC to cause the same code that is generated for one instance to be shared or used by another instance. In other words, if a pragma SHARE_GENERIC applies to a generic declaration or to a specific instance, the compiler tries to generate code that can be shared by subsequent instantiations of the same generic. Chapter 12 of the *VAX Ada Language Reference Manual* gives the syntax and rules for using the pragma SHARE_GENERIC. For convenience, the syntax is summarized here, as follows:

```
pragma SHARE_GENERIC (name {,name});
```

For example:

```
generic
   type INPUT_TYPE is private;
procedure USE_OFTEN (X: INPUT_TYPE);
pragma SHARE_GENERIC(USE_OFTEN);

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

procedure USE_OFTEN (X: INPUT_TYPE) is
begin
   null;
end USE_OFTEN;
```

```
with USE_OFTEN;
package USE_SHARE_GENERIC is

   procedure USE_INTEGER is new USE_OFTEN(INTEGER);
   procedure USE_FLOAT is new USE_OFTEN(FLOAT);
   procedure USE_STRING is new USE_OFTEN(BOOLEAN);

end USE_SHARE_GENERIC;
```

In this example, the compiler generates shareable code for each of the instantiations declared in the package USE_SHARE_GENERIC.

The code generated for one instance cannot be shared by another instance unless you specify a pragma SHARE_GENERIC for the instance or for the generic from which the instance was generated. You can also control generic code sharing with the /OPTIMIZE qualifier at compile time (see *Developing Ada Programs on VMS Systems*).

Generic code sharing provides four important benefits:

- It saves compilation time when the generated machine code would otherwise be large. When generic code sharing is in effect, the compiler does not have to generate code for each instantiation.
- It makes a program significantly smaller when generic instantiations would otherwise generate a large amount of machine code or a large number of constants.
- It gives Ada programmers the engineering advantages of a strongly typed language without consuming extra memory for multiple copies of essentially identical algorithms.

  In a strongly typed language, such as Ada, a program often uses generics to define operations such as mathematical functions, sorting, symbol table management, list management, and so on. The program instantiates these generics to provide the same operations on a variety of types.

  In an older language that does not support or encourage the use of large numbers of types (for example, FORTRAN, C, BLISS, or assembly language), such operations would have been written as one piece of code, and then "shared" among the various types. For example the VMS Run-Time Library routine MTH$SQRT would be used to take the square root of an F_floating variable regardless of whether that variable was logically meters-per-second or kilograms-per-square-meter. Similarly, to write an input-output subsystem where files of many different types needed to be supported, C or BLISS programmers would use common code to which just the address and the length of the data were passed. The same piece of code could be used to manipulate the data, regardless

of the fact that one piece of data would be a *file-of*-STRING, another a *file-of*-COMPLEX_NUMBER, and so on. In effect, the programmer was organizing the sharing of the machine instructions to avoid having multiple copies of essentially identical code.

Before generic code sharing was available, writing generics for operations such as list management saved rewriting the same algorithms many times, but did not save the amount of code that was generated. Thus, generic code sharing eliminates one of the possible advantages of a less strongly typed language over Ada.

- It allows Ada programmers to write subprograms with call-back routines, again, without incurring the penalty of generating duplicate code.

  Another feature of older languages is the ability to provide a "call-back routine" as a variable or parameter, so that an algorithm can be tailored by its caller. For example, a SORT routine might accept COMPARE and EXCHANGE routines as parameters. The SORT routine executes the general flow of the sorting algorithm, but calls back to the specific COMPARE and EXCHANGE routines to achieve a particular kind of sort.

  In Ada, you pass subprograms as parameters by declaring them as formal subprogram parameters in generic declarations. For example:

```
generic

   type ITEM is private;

   type VECTOR is array (NATURAL range <>) of ITEM;

   with function "<"(LEFT, RIGHT : ITEM) return BOOLEAN;
   with procedure EXCHANGE(LEFT, RIGHT : in out ITEM);

procedure GENERIC_SORT(LIST : VECTOR) is
begin
    . . .
end;
```

  Without shared generics, this approach causes extra memory to be consumed for each instantiation; duplicate code is generated, even though the only difference between instantiations may be the call-back routines. Generic code sharing reintroduces this ability into Ada without any penalties.

You can maximize generic code sharing either by specifying the /OPTIMIZE=SHARE:MAXIMAL qualifier at compile time or by specifying a pragma SHARE_GENERIC for all generic declarations and/or instantiations in your application. In some cases, maximal generic code sharing can result in a dramatic decrease in the size of your program and can greatly improve

run-time performance (particularly elapsed time). However, the benefits of maximal sharing depend on the characteristics of your application. Often you can obtain the best results by specifying the pragma SHARE_GENERIC for particular generic declarations and/or instantiations rather than compiling all units with the /OPTIMIZE=SHARE:MAXIMAL qualifier.

Generic code sharing is intended to reduce the size of your program. Shared code generally executes more slowly than nonshared code because sharing adds some processing overhead and prevents optimizations that are based on the actual parameters provided for a particular instantiation. However, generic code sharing will occur only if the code that is generated for one instance is similar to the code generated for another. Thus, the execution times for shared code are often similar to those for nonshared code, particularly for larger generic packages.

Note that although generic code sharing is intended to reduce the size of your program, it can increase program size under some conditions. For example:

- Shared code for one instance is always larger than the corresponding nonshared code. Thus, the size of your program will increase if shareable code is generated for one instance, but is never shared by another.
- Sharing can also increase the size of your program if generics are instantiated a relatively small number of times or if the actual parameters for each instantiation of a particular generic are sufficiently different to preclude sharing.

Sharing the code for two instantiations of a large generic reduces the size of your program, but you may need to share the code for many instantiations of a smaller generic to achieve a net reduction in program size.

9.3.3 Library-Level Generic Instantiations

If you have a program that makes multiple instantiations of the same generic, you can save compile time and often make your program more efficient by first creating a library package that instantiates the generic and then making that package available to your program (by using **with** and **use** clauses).

For example, suppose that you have defined a package containing a floating-point type and operations on that type. Also suppose that you want to be able to include the predefined VAX Ada mathematics functions (in the generic package MATH_LIB) as operations, and you want to be able to use TEXT_IO operations to perform input and output. The most efficient way of making your type, its operations, and the instantiations of MATH_LIB and

TEXT_IO.FLOAT_IO available to your program is to make a library package as follows:

```
with TEXT_IO; use TEXT_IO;
with MATH_LIB;
package MY_FLOAT_TYPE_OPS is
   type MY_FLOAT is digits 13;
   package MY_FLOAT_TEXT_IO is new FLOAT_IO(MY_FLOAT);
   package MY_FLOAT_MATH_LIB is new MATH_LIB(MY_FLOAT);
    . . .
end MY_FLOAT_TYPE_OPS;
```

When you make this package available to your program (or to parts of your program), the instantiations of TEXT_IO.FLOAT_IO and MATH_LIB will be done only once (when the package is initially compiled and added to your program library), not each time you use them.

VAX Ada supplies a set of predefined library packages that instantiate commonly used generics, notably the generic TEXT_IO packages for integer and floating-point input and output, and the generic package MATH_LIB for floating-point mathematical operations. (See Chapter 3, Chapter 6, and Appendix B for the descriptions and specifications of these predefined packages.)

For example, if you needed to use the operations in MATH_LIB and TEXT_IO.FLOAT_IO many times throughout your program on objects of the type LONG_FLOAT, you could use the appropriate predefined packages, as follows, to save compile time and object code size:

```
with LONG_FLOAT_TEXT_IO;  use LONG_FLOAT_TEXT_IO;
with LONG_FLOAT_MATH_LIB; use LONG_FLOAT_MATH_LIB;
procedure MY_MAIN is
   X: LONG_FLOAT;
begin
 . . .
   PUT(SIN(X));
 . . .
end MY_MAIN;
```

The instantiations of TEXT_IO.FLOAT_IO and MATH_LIB for the type LONG_FLOAT are done once, but are available at all levels of MY_MAIN.

9.4 Techniques for Reducing CPU Time and Elapsed Time

You can use a variety of techniques to significantly reduce the CPU time and elapsed time required to execute a VAX Ada program on a VMS system.

To decrease the program's CPU time on a particular VAX processor, you can make three basic changes to the program:

- Decrease the number of instructions being executed.
- Decrease the number of expensive instructions being executed.
- Decrease the amount of data being read from and written to memory.

To decrease the program's elapsed time, you can also make three basic changes to the program:

- Decrease the CPU time.
- Decrease the amount of time spent waiting for input-output and page faults.
- Overlap the CPU time with the time spent waiting for input-output.

The following sections discuss these changes and some of the techniques for making them.

NOTE

The VAX Performance and Coverage Analyzer (PCA) is an optional layered product, which is also included among the VAXset tools. It measures the performance characteristics of user-mode programs running on VMS systems. While the following discussions may offer some general assistance, the techniques they propose are best used in conjunction with VAX PCA.

To use VAX PCA, you must have it installed on your system, and you must compile and link your program with /DEBUG qualifiers in effect. See the VAX PCA documentation for information on how to use VAX PCA.

9.4.1 Decreasing the CPU Time of a VAX Ada Program

The first step in decreasing the CPU time of a VAX Ada program is to use VAX PCA to identify the parts of the program that are using the most CPU time.

In the parts of the program that do not use significant amounts of CPU time, you will not gain much of a performance improvement by suppressing checks, explicitly expanding subprograms inline, or otherwise writing anything other than straightforward Ada code. Thus, you should also first compile your program without the effects of the pragmas INLINE, SUPPRESS, or SUPPRESS_ALL. You can use the /OPTIMIZE=(INLINE:NONE) and /CHECK compilation qualifiers to cause the compiler to ignore any INLINE and SUPPRESS pragmas that are already in your source files.

Once you have identified the part of the program that uses most of the CPU time, you should next evaluate the algorithms that you have used in that part. Wherever possible, you should replace the algorithms with significantly more efficient algorithms or you should use more efficient data structures. For example, if the algorithm in question is an expensive calculation, you may be able to replace it with some form of table lookup. Furthermore, you may be able to reorganize the program as a whole to decrease the number of times the expensive algorithms are executed.

Once you have implemented the most efficient algorithms, the next step is to decrease the number of instructions executed in places where significant amounts of CPU time are being used. There are some techniques that you can use to significantly decrease the number of instructions. These techniques are discussed in the following sections.

NOTE

Because these techniques involve changing code (often converting small pieces of code into larger and more complex forms), you should use them only in the parts of the program that would really benefit. Again, VAX PCA can help you correctly identify the parts of the program that you should consider rewriting.

Before you rewrite your Ada code, examine the machine code produced by the compiler to determine if any improvement is possible. You can examine the machine code either by stepping through it using the VMS Debugger or by examining a listing file that you have produced with the /LIST/MACHINE_CODE qualifier at compile time.

9.4.1.1 Eliminating Run-Time Checks

Run-time checks are the easiest of all overhead checks to eliminate. You can eliminate run-time checks completely with the pragma SUPPRESS_ALL. However, eliminating checks in this way is not safe: an error condition that would trigger a check may still occur (for example, a null access value is deaccessed, an array is indexed outside of its bounds, and so on). Instead, you should write your program so that the compiler can deduce reliably that a check would never be triggered, and code would not be generated for the check. For example:

- Use subranges so that range checks are removed from loops.

  The compiler uses extensive knowledge of subtypes to eliminate checks or move them out of loops. If the compiler has not deduced that a check either does not need to be done or can be moved out of a loop, you can give it extra clues by defining subtypes outside of the loop. The compiler will then perform the check outside of the loop, and will use the information it gains inside of the loop to eliminate a check.

  For example, the following code causes a check to be done inside a loop:

```
procedure ZERO( N, M         : POSITIVE;
                ARRAY_PARAMETER : in out ARRAY_TYPE) is
begin
   for I in N..M loop
      ARRAY_PARAMETER(I) := 0;  -- Check needed inside.
   end loop;
end;
```

  This is a more efficient version:

```
procedure ZERO( N, M         : POSITIVE;
                ARRAY_PARAMETER : in out ARRAY_TYPE) is
   subtype S is
   INTEGER range ARRAY_PARAMETER'FIRST..ARRAY_PARAMETER'LAST;

begin
   for I in S range N..M loop   -- Check done here, outside the
      ARRAY_PARAMETER(I) := 0;  -- loop, so no check needed
                                -- inside.
   end loop;
end;
```

- Use renaming to remove checks from loops.

  Names involving the following constructs require checks to determine if the exception CONSTRAINT_ERROR should be raised (see Chapter 4 of the *VAX Ada Language Reference Manual*):

  - An access value used as a prefix (for example, A.**all**)
  - Indexing (for example, A(23)) or slicing (for example, A(1..10))

- Selecting a component of a variant part of a record (for example, A.C)

Usually the compiler detects such names as being loop-invariant, and moves them out of the loop. If the machine code indicates that this optimization has not happened, you can use a renaming outside of the loop to move the checking code outside of the loop.

For example, the following block does not do an INDEX_CHECK each time the loop is executed:

```
declare
   COMPONENT : POSITIVE renames ARRAY_OF_POSITIVE(I-4);
begin
   loop
      COMPONENT := {expression};
      ...
   end loop;
end;
```

You can also use this technique to avoid repeated checks in code that has no loops.

9.4.1.2 Reducing Function and Procedure Call Costs

You can reduce function and procedure call costs with the following techniques:

- Use the pragma INLINE to eliminate call and return overhead for calls of trivial subprograms. For larger subprograms, this technique helps only if the inline-expanded version of the subprogram can then be significantly optimized. This effect often happens when one of the actual parameters is a constant.

  The compiler automatically expands some subprograms inline, but it cannot do so if extra dependences are created. The pragma INLINE gives the compiler permission to add these dependences. The pragma INLINE also forces the compiler to expand calls inline when it would have otherwise decided that the inline expansion was not worthwhile.
- Use the /OPTIMIZE=INLINE:SUBPROGRAMS or /OPTIMIZE=INLINE:MAXIMAL compilation qualifier to direct the compiler to eliminate call and return overhead for calls to trivial subprograms in other units. See *Developing Ada Programs on VMS Systems* for more information on the /OPTIMIZE qualifier.
- Use the pragma ELABORATE to eliminate access-before-elaboration checks on subprograms that have been expanded inline.

For very small subprograms in other units, the cost of the run-time check to see if the subprogram body has been elaborated may be significant. The pragma ELABORATE provides a way of forcing the elaboration order, and the compiler uses this knowledge to eliminate the check.

For example:

```
-- First compilation unit.
--
package PKG is
function TRIVIAL return INTEGER;
pragma INLINE(TRIVIAL);
end;

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

-- Second compilation unit.
--
package body PKG is
   I : INTEGER := 0;
   function TRIVIAL return INTEGER is
   begin
      I := I+1;
      return I;
   end;
end;

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

-- Third compilation unit.
--
with PKG;
pragma ELABORATE(PKG);
procedure EXAMPLE is
   J : INTEGER := PKG.TRIVIAL;  -- Pragma ELABORATE guarantees
                                -- that TRIVIAL's body must have
                                -- been elaborated, so no check
                                -- is needed.
begin
   null;
end;
```

- Use records to pass multiple parameters quickly, and to move the evaluation of parameters to less frequently executed regions of the code.

  For example, the procedure EXAMPLE in the following code incurs some run-time overhead when it makes the call to PKG.PROC, because of the number of parameters and parameter evaluations:

```
-- First compilation unit.
--
package PKG is
   procedure PROC(P1, P2 : INTEGER; P3, P4 : FLOAT; P5 : BOOLEAN);
end;

-- Second compilation unit.
--
with PKG;
procedure EXAMPLE is
begin
   for I in 1..10 loop
       . . .
      PKG.PROC(1, I, 0.0, FLOAT(I)*3.0, FALSE);
       . . .
   end loop;
end;
```

This example would run more efficiently if it were rewritten as follows:

```
-- First compilation unit.
--
package PKG is
   type PROC_PARAMETERS is
      record
         P1, P2 : INTEGER;
         P3, P4 : FLOAT;
         P5 : BOOLEAN;
      end record;
   procedure PROC(P : PROC_PARAMETERS);
end;

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

-- Second compilation unit.
--
with PKG;
procedure EXAMPLE is
   P : PKG.PROC_PARAMETERS;
begin
   P.P1 := 1;        -- Notice that the cost of setting up
   P.P3 := 0.0;      -- these parameters has been moved out of
   P.P5 := FALSE;    -- the loop...
      for I in 1..10 loop
       . . .
      P.P2 := I;
      P.P4 := FLOAT(I)*3.0;
      PKG.PROC(P);      -- And that it requires fewer
       . . .            -- instructions to pass just
                        -- one parameter.
   end loop;
end;
```

9.4.1.3 Using Scalar Variables and Avoiding Expensive Operations on Composite Types

In general, the current state of optimizing compilers is such that they are much better at generating code for operations involving simple types than they are at generating code for operations involving composite types. For this reason, and because of slight differences in the results if exceptions occur, the following changes may make a significant difference in frequently executed code:

- Replace complex operations on composite types with a series of simpler operations, especially if the result can be assigned directly into its final place.

  For example, replace an assignment like this:

  ```
  A := B & (1..A'LENGTH - B'LENGTH => ' ');
  ```

  With the following operations:

  ```
  A(1..B'LENGTH) := B;
  for I in A'LENGTH + 1..B'LENGTH loop
        A(I) := ' ';
  end loop;
  ```

- Rather than using aggregates, especially those involving run-time expressions, build values in place.

  For example, replace this single operation:

  ```
  A := (I*I, 2*J, K+0.3);
  ```

  With this series of smaller operations:

  ```
  A.C1 := I*I;
  A.C2 := 2*J;
  A.C3 := K+0.3;
  ```

- Sometimes it pays to pull components out into a scalar constant, so that the compiler knows that various values are not modified by assignments to other components.

  For example, an examination of the machine code for the following Ada code may show that V.C is being repeatedly fetched from memory:

  ```
  A.C1 := A.C1*V.C;
  A.C2 := A.C2+V.C;
  A.C3 := A.C3/V.C;
  ```

If that is true, you should replace the Ada code with something like this:

```
declare
   X : constant FLOAT := V.C;
begin
   A.C1 := A.C1*X;
   A.C2 := A.C2+X;
   A.C3 := A.C3/X;
end;
```

- Use access-to-composite types rather than returning large composite objects as values.

  For example, code like this:

```
package AIRPLANE_INFO_PKG is

   . . .
   type AIRPLANE_INFO_TYPE is
      record
         WEIGHT : KILOGRAMS;
         . . .
      end record;

   function GET_AIRPLANE_INFO(NAME : STRING)
      return AIRPLANE_INFO_TYPE;

end;
```

  Should be replaced with code like this:

```
package AIRPLANE_INFO_PKG is

   . . .
   type AIRPLANE_INFO_TYPE is
      record
         WEIGHT : KILOGRAMS;
         . . .
      end record;

   type ACCESS_AIRPLANE_INFO_TYPE is
      access AIRPLANE_INFO_TYPE;
   function GET_AIRPLANE_INFO(NAME : STRING)
      return ACCESS_AIRPLANE_INFO_TYPE;

end;
```

- Use **in** or **in out** parameters to allow the compiler to assign values directly to target variables, rather than making assignments with function results. For example:

```
package VECTOR_PKG is

   type VECTOR is
      record
         I, J, K : FLOAT;
      end record;

   -- Provide all three forms of ADD, so that the caller
   -- can choose the most efficient.
   --
   function "+"(LEFT, RIGHT : VECTOR) return VECTOR;

   procedure ADD(LEFT : VECTOR; RIGHT : in out VECTOR);

   procedure ADD(LEFT, RIGHT : VECTOR; RESULT : out VECTOR);

end;

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

with VECTOR_PKG; use VECTOR_PKG;
procedure EXAMPLE(A, B : VECTOR; R : out VECTOR) is
begin
   R := A + B;    -- Less efficient.
   ADD(A, B, R); -- More efficient.
end;
```

9.4.2 Decreasing the Elapsed Time of a VAX Ada Program

Elapsed time is a consequence of time spent executing instructions, paging, and doing input-output. You may be able to decrease the instruction execution time as described in Section 9.4.1. Once you have done that, the only alternatives are to obtain either a faster CPU or more CPUs. You should wait to explore these last two alternatives until you have examined the program's paging and input-output behavior. The following sections discuss paging and input-output effects in more detail.

Note the following information about using different or more CPUs:

- If you obtain a faster CPU, your program's run-time performance will improve just by running the program if the elapsed time was spent executing instructions, rather than waiting for input-output.
- If you have chosen to use more than one CPU to improve performance, then you should consider breaking your single VAX Ada program into multiple VAX Ada programs, and then using either networking or shared global sections to communicate the data between them.

  Chapter 10 includes a section with an example program that shares memory between one or more CPUs on a VMS system.

9.4.2.1 Controlling Paging Behavior

Experience has shown that, in general, the VMS Linker and image activator do an excellent job of controlling the paging of a program's instructions. The most likely cause of excessive paging is having an insufficient working set or processing the data in a "jump-around" manner.

A solution to the working-set problem is either to increase the working set size, or to design your program so that it handles its data in "working-set-sized" pieces. The latter solution is very difficult to apply to existing code.

The worst examples of jumping around are caused when large multidimensional arrays are accessed so that the first index changes the fastest. Note that this effect occurs in an opposite way in FORTRAN, where it is desirable to change the first index the fastest.

9.4.2.2 Improving Input-Output Behavior

Input-output is usually bounded by the device you are using. You can gain improvements by taking one of the following actions:

- Reading or writing more data to the device in a single operation
- Packing the types involved, so that fewer bytes are needed for the values
- Using a faster device

You can also gain significant improvements by calling asynchronous input-output routines (RMS and system service routines), and starting read requests some time before the data being obtained is actually needed. See the *VMS Record Management Services Manual* for more information on RMS routines; see the *VMS System Services Volume* for more information on VMS system service routines.

9.4.2.3 Overlapping Unrelated Input-Output and Instruction Execution

An application can sometimes exploit Ada multitasking to overlap the time spent waiting for an input-output operation with some computation. You can achieve this effect by putting the input-output in a high-priority task and the computation in a low-priority task. The difference in priorities is required so that the input-output-bound task will access the CPU of the computing task, get its next input-output started, and then wait—thus returning control to the computing task. If the computing task is given the higher, or even the same, priority, then the input-output-bound task will not be able to start its input-output as soon as possible, and thus its elapsed time will be extended.

For example:

```
-- A high-priority task to drive a graphics device at full speed.
--
task GRAPHICS_ENGINE is
   pragma PRIORITY (8);
   entry PUT_PICTURE(P : ACCESS_PICTURE);
end;

task body GRAPHICS_ENGINE is
   P : ACCESS_PICTURE;
begin
   loop
      accept PUT_PICTURE(P : ACCESS_PICTURE) do
         GRAPHICS_ENGINE.P := P;
      end;

      DRAW(P);              -- Draw to a hidden plane of graphics
                            -- memory.  The device takes a while to
                            -- do this, but returns immediately.

      delay 0.1;            -- Give the device a chance to draw
                            -- the picture.

      FLIP_VISIBLE;         -- Make the hidden plane visible and the
                            -- old plane invisible.

      ERASE;                -- Erase the hidden plane of graphics.
                            -- Again, the device takes a while, but
                            -- returns immediately.
   end loop;
end;

-- A lower-priority task to decide what to draw.
--
task GENERATE_PICTURE is
   pragma PRIORITY (7);
end;

task body GENERATE_PICTURE is
begin
   loop
      -- {compute a new picture};
      GRAPHICS_ENGINE.PUT_PICTURE(P);
   end loop;
end;
```

Most of the computation in this example will be done while the GRAPHICS_ENGINE is executing its **delay** statement; if 0.1 second is sufficient time to do all of the computation, the device will be driven at full speed.

Chapter 10

Additional Programming Considerations

This chapter documents VAX Ada programming considerations that may not be immediately obvious, but that may affect the run-time behavior or performance of your VAX Ada programs. It also documents the use of some of the low-level, system-specific features of VAX Ada.

10.1 Working with Address Values

To allow you to work with storage addresses, the Ada language provides the predefined type ADDRESS (in the package SYSTEM) and the ADDRESS attribute. To avoid difficult-to-isolate problems when working with values of this type or values returned by this attribute in VAX Ada, make sure that you do not use them to do any of the following:

- Reference an object whose lifetime has expired.
- Reference an object in an inappropriate manner (for example, you try to change a declared constant, or the value of an **in** parameter).
- Access storage beyond the end of the amount allocated for an object.
- Access a variable by more than one path, unless that variable has been declared with the pragma VOLATILE.
- Place a value into a variable that is inconsistent with the variable's declared type or subtype.

If a subprogram body, task body, or library package elaboration code uses the ADDRESS attribute of an **out** or **in out** formal parameter, or of a variable whose declaration does not include a pragma VOLATILE, the VAX Ada compiler will implicitly treat that parameter or variable as being locally volatile. (Being locally volatile means being volatile for all of the

immediate block statement, body, or library package elaboration code; not for any surrounding or enclosed subprogram bodies, tasks, or library package elaboration code.)

The effect of this rule is to suppress optimizations that assume the compiler can detect all changes to the value of the parameter or variable.

For example, the statements in the following procedure leave X with a value of 0, rather than 1. If X is not implicitly treated as being locally volatile (because of the use of X'ADDRESS), the optimizer may generate code using the most recent assignment, or 1, when it assigns the value of X to Y. However, because X has been marked as being locally volatile, the optimizer instead generates code that causes the value at the address of X, in this case 0, to be retrieved when the assignment to Y is made.

```
with SYSTEM; use SYSTEM;
with UNCHECKED_CONVERSION;
procedure SHOW_CONVERT is

   type ACCESS_INTEGER is access INTEGER;

   function CONVERT_ADDRESS_TO_ACCESS_INTEGER is
      new UNCHECKED_CONVERSION (ADDRESS, ACCESS_INTEGER);

   X, Y  : INTEGER;
   V1    : ADDRESS;
   V2    : ACCESS_INTEGER;

begin
 . . .
   V1 := X'ADDRESS;
   V2 := CONVERT_ADDRESS_TO_ACCESS_INTEGER(V1);
   X := 1;
   V2.all := 0;      -- X is now 0,
   Y := X;           -- so Y is 0.
 . . .
end SHOW_CONVERT;
```

10.2 Using Low-Level System Features

The predefined package SYSTEM provides a number of useful, low-level type declarations and operations. The following sections give advice on using these declarations and operations, and, where appropriate, provide some examples of possible applications.

10.2.1 The VAX Device and Processor Register and Interlocked Operations

Applications accessing VMS input-output space or using shared memory have special restrictions on which VAX instructions can be used. You can force the VAX Ada compiler to meet these restrictions by using the following operations defined in the package SYSTEM:

Operation	Equivalent VAX Instruction
Function READ_REGISTER	—
Function WRITE_REGISTER	—
Function MFPR	Move from Process Register (MFPR)
Procedure MTPR	Move to Process Register (MPTR)
Procedure CLEAR_INTERLOCKED	Branch on Bit Clear and Clear Interlocked (BBCCI)
Procedure SET_INTERLOCKED	Branch on Bit Set and Set Interlocked (BBSSI)
Procedure ADD_INTERLOCKED	Add Aligned Word Interlocked (ADAWI)
Procedure INSQHI	Insert Entry into Queue at Head (INSQHI)
Procedure REMQHI	Remove Entry from Queue at Head (REMQHI)
Procedure INSQTI	Insert Entry from Queue at Tail (INSQTI)
Procedure REMQTI	Remove Entry from Queue at Tail (REMQTI)

The *VAX Ada Language Reference Manual* specifies and gives the syntax for these operations. The *VAX Architecture Reference Manual* and *VAX Hardware Handbook* give detailed information on the VAX instructions themselves.

Example 10–1 shows one method of implementing a queue using the interlocked queue operations. Note that the interlocked queue operations all require quadword alignment of the queue elements. To satisfy this requirement, you can use Ada alignment clauses. However, remember that VAX Ada has some allowed restrictions on the alignments that you can specify in alignment clauses. In particular, the maximum alignment for stack-allocated record objects is a longword.

See the *VAX Ada Language Reference Manual* for more information on the alignment clauses and the allowed restrictions. See Chapter 2 for additional information on the use of alignment clauses and for information on how and where storage for objects is allocated.

Example 10–1: One Use of the Interlocked Queue Operations

```
package DEFINE_DYN_QUEUE is
   type FORWARD_BACKWARD is
      record
         FORWARD,BACKWARD: INTEGER := 0;
      end record;
   for FORWARD_BACKWARD use record at mod 8;
         FORWARD at 0 range 0..31;
         BACKWARD at 4 range 0..31;
      end record;
   for FORWARD_BACKWARD'SIZE use 64;

   type R is
      record
         FB: FORWARD_BACKWARD;
         VALUE: INTEGER;
      end record;
   for R use
      record
         FB at 0 range 0..63;
      end record;

   type H_PTR is access FORWARD_BACKWARD;
   type Q_PTR is access R;

end DEFINE_DYN_QUEUE;
------------------------------------------------------------------------
with SYSTEM; use SYSTEM;
with DEFINE_DYN_QUEUE; use DEFINE_DYN_QUEUE;
with UNCHECKED_CONVERSION;
procedure DYNAMIC_QUEUE is
   --
   -- This procedure does nothing more than create an interlocked
   -- queue, add entries to the head and tail, and then delete
   -- the queue by removing the entries. The queue head and
   -- elements are defined as access types (declared in the
   -- package DEFINE_DYN_QUEUE). An alternative means of
   -- implementing the queue would be to declare a static set of
   -- record type elements (instead of access type elements) in
   -- the package DEFINE_DYN_QUEUE, and then create or delete the
   -- queue.
   --

   -- Example of a conversion function for converting from
   -- addresses to access types (not used in this program).
   --
   function ADDR_TO_ACCESS_R is
      new UNCHECKED_CONVERSION(ADDRESS,Q_PTR);
```

(continued on next page)

Example 10–1 (Cont.): One Use of the Interlocked Queue Operations

```
   -- Define variables for use with the interlocked queue
   -- operations.
   --
   IN_STATUS: INSQ_STATUS;
   REM_STATUS: REMQ_STATUS;
   OUT_ADDRESS: ADDRESS;

   -- Define queue head and element variables for use in
   -- constructing the queue.
   --
   HEAD: H_PTR := new FORWARD_BACKWARD;
   ELEMENT: Q_PTR;

begin

   -- Given the head (HEAD), create some elements and insert them
   -- at the head (INSQHI) and tail (INSQTI).
   --
   ELEMENT := new R;
   ELEMENT.VALUE := 1;
   INSQHI (ITEM   => ELEMENT.all'ADDRESS,
           HEADER => HEAD.all'ADDRESS,
           STATUS => IN_STATUS);
   ELEMENT := new R;
   ELEMENT.VALUE := 2;
   INSQHI (ITEM   => ELEMENT.all'ADDRESS,
           HEADER => HEAD.all'ADDRESS,
           STATUS => IN_STATUS);
   ELEMENT := new R;
   ELEMENT.VALUE := 3;
   INSQTI (ITEM   => ELEMENT.all'ADDRESS,
           HEADER => HEAD.all'ADDRESS,
           STATUS => IN_STATUS);
   ELEMENT := new R;
   ELEMENT.VALUE := 4;
   INSQTI (ITEM   => ELEMENT.all'ADDRESS,
           HEADER => HEAD.all'ADDRESS,
           STATUS => IN_STATUS);

   -- Now, remove all the elements from the queue.
   --
   REMQHI (HEADER => HEAD.all'ADDRESS,
           ITEM   => OUT_ADDRESS,
           STATUS => REM_STATUS);
   REMQHI (HEADER => HEAD.all'ADDRESS,
           ITEM   => OUT_ADDRESS,
           STATUS => REM_STATUS);
```

(continued on next page)

Example 10–1 (Cont.): One Use of the Interlocked Queue Operations

```
   REMQTI (HEADER => HEAD.all'ADDRESS,
           ITEM   => OUT_ADDRESS,
           STATUS => REM_STATUS);
   REMQTI (HEADER => HEAD.all'ADDRESS,
           ITEM   => OUT_ADDRESS,
           STATUS => REM_STATUS);

end DYNAMIC_QUEUE;
```

10.2.2 Unsigned Types in the Package SYSTEM

When working with the unsigned types declared in the package SYSTEM, note that they have the following ranges:

UNSIGNED_BYTE	0..255 0..16#FF#
UNSIGNED_WORD	0..65535 0..16#FFFF#
UNSIGNED_LONGWORD	–2,147,483,648..2,147,483,647 ($-2^{31}..2^{31} - 1$ or MIN_INT..MAX_INT) –16#80000000#..16#7FFFFFFF#

Note in particular that the type UNSIGNED_LONGWORD is really a signed type; its range is MIN_INT..MAX_INT, not 0..2137483647 (0..16#FFFFFFFF#). The choice of range and representation for the type SYSTEM.UNSIGNED_LONGWORD is based on the following constraints:

- VMS system routines often require that unsigned longwords be of an integer type, rather than, for example, an array of BOOLEAN components.
- The Ada language requires that integer types be symmetric about 0.
- The VAX hardware poses difficulties for operations on an integer type larger than 32 bits.

For example, consider the expected declaration:

```
type UNSIGNED_LONGWORD is range 0..2**32-1;
```

According to the Ada language rules, this declaration is equivalent to the following declarations:

```
type UNSIGNED_LONGWORD_type is new predefined_integer_type;
subtype UNSIGNED_LONGWORD is UNSIGNED_LONGWORD_type range 0..2**32-1;
```

Because the Ada language requires that predefined integer types be symmetric about 0, the predefined type chosen for UNSIGNED_LONGWORD must have at least the following range:

```
type predefined_integer_type is range -(2**32)..2**32-1;
```

This symmetry is required because the language specifies that the predefined operations on integer types can raise the exception NUMERIC_ERROR or CONSTRAINT_ERROR only if the result is not a value of the predefined type (see Chapter 4 of the *VAX Ada Language Reference Manual*). If you were to declare variables A, B, C, and D to be of the type UNSIGNED_LONGWORD in the preceding example, then an assignment statement like the following would raise an exception because it involves intermediate negative calculations (B – C), which are not values of the predefined type:

```
A := B - C + D;
```

If this definition of UNSIGNED_LONGWORD were represented with the expected representation (0..16#FFFFFFFF#), then operations involving negative intermediate results would have to account for at least the value $-2^{32} - 1$. The need for an extra sign bit would cause UNSIGNED_LONGWORD operations to be carried out as quadword operations (a 32-bit longword is one bit too small to handle the value $-2^{32} - 1$). Because the VAX hardware does not support all arithmetic operations on quadwords, this implementation would be inefficient. So the VAX Ada implementation defines UNSIGNED_LONGWORD as if it were an integer type, with the range MIN_INT..MAX_INT ($-2^{31}..2^{31} - 1$) and the representation (16#80000000#..16#7FFFFFF#).

A similar analysis applies to SYSTEM.UNSIGNED_WORD, although the inefficiency of the implementation is not as high as for UNSIGNED_WORD because longword instructions can be used for the operations on the type. Thus, the definition of UNSIGNED_WORD is the expected definition (0..65535).

An alternative to the type SYSTEM.UNSIGNED_LONGWORD is the type SYSTEM.BIT_ARRAY_32, which is a 32-bit array type with components of the type STANDARD.BOOLEAN. The package SYSTEM also provides conversion functions so that you can convert values of the type SYSTEM.UNSIGNED_LONGWORD to the type SYSTEM.BIT_ARRAY_32 and vice versa. For example, you can define your unsigned longword variables using the type SYSTEM.BIT_ARRAY_32 and then convert them to the

type SYSTEM.UNSIGNED_LONGWORD using the function SYSTEM.TO_UNSIGNED_LONGWORD. For example:

```
with SYSTEM;
with TEXT_IO; use TEXT_IO;
procedure USE_UNSIGNED_LONGWORD is

   package UL_TEXT_IO is new INTEGER_IO(SYSTEM.UNSIGNED_LONGWORD);
   use UL_TEXT_IO;

   BASE_16: NUMBER_BASE := 16;
   OUTPUT: SYSTEM.UNSIGNED_LONGWORD := 0;
   VAR1, VAR2, RESULT: SYSTEM.BIT_ARRAY_32;

begin

   VAR1 := SYSTEM.BIT_ARRAY_32'(0..2   => TRUE,
                                30..31 => TRUE,
                                others => FALSE);
   OUTPUT := SYSTEM.TO_UNSIGNED_LONGWORD(VAR1);
   PUT(ITEM => OUTPUT,
       BASE => BASE_16);
   NEW_LINE;

   VAR2 := SYSTEM.BIT_ARRAY_32'(0      => TRUE,
                                others => FALSE);
   OUTPUT := SYSTEM.TO_UNSIGNED_LONGWORD(VAR2);
   PUT(ITEM => OUTPUT,
       BASE => BASE_16);
   NEW_LINE;

   RESULT := SYSTEM."xor"(VAR1,VAR2);
   OUTPUT := SYSTEM.TO_UNSIGNED_LONGWORD(RESULT);
   PUT(ITEM => OUTPUT,
       BASE => BASE_16);

end USE_UNSIGNED_LONGWORD;
```

When you are working with SYSTEM.UNSIGNED_LONGWORD, note also that the following declaration will raise the exception CONSTRAINT_ERROR:

```
X : SYSTEM.UNSIGNED_LONGWORD := 16#80000000#;
```

Recall that –1 is not a literal; it is an expression consisting of a unary adding operator followed by a decimal literal. 16#80000000# is the decimal literal representing 2^{31}, which is 1 greater than UNSIGNED_LONGWORD'LAST. Thus, CONSTRAINT_ERROR is raised.

If you need to work with "unsigned" numbers with this particular bit pattern or a pattern similar to it, use negative numbers, or the FIRST attribute. For example, the following declarations and assignments will not raise CONSTRAINT_ERROR:

```
with SYSTEM;
procedure TRY_LONGWORD is
   A: constant := -16#80000000#;
   B: SYSTEM.UNSIGNED_LONGWORD := A;
   C: INTEGER := A;
begin
   . . .
end TRY_LONGWORD;
```

10.3 Working with Varying Strings

Because Ada does not have a predefined varying string type, you must use a record or an access type to declare a varying string in Ada. When working with varying strings, be aware that one of the most common ways to raise the exception CONSTRAINT_ERROR occurs in the following case:

```
subtype STRING_SIZE is NATURAL range 0..NATURAL'LAST;
type V_STRING (SIZE : STRING_SIZE := 0) is
   record
      L : STRING_SIZE;
      S : STRING(1..SIZE);
   end record; -- The maximum size of records of this type
               -- (NATURAL'LAST + 8) is a number that is greater
               -- than the largest value that can be computed
               -- in a VAX 32-bit integer (NATURAL'LAST).

X: V_STRING;    -- Unconstrained object.
```

In this case, X is an unconstrained object, to which any value of the type V_STRING can be assigned. For such objects, the Ada language standard permits an implementation to allocate the maximum size required for any value of the type at the time the object is elaborated; VAX Ada, in fact, does just this. So, when X is elaborated, the compiler tries to allocate space for the largest object of the type V_STRING. First, the compiler computes the maximum size for an object of the type. This computation, like any integer computation in VAX Ada, must not exceed the implementation-defined limit for type INTEGER (SYSTEM.MAX_INT, or $2^{31} - 1$; otherwise, CONSTRAINT_ERROR must be raised (see Chapter 11 and Appendix F of the *VAX Ada Language Reference Manual*).

For the type V_STRING, the component S requires up to $2^{31} - 1$ bytes, the component L requires another 4 bytes, and the discriminant SIZE requires another 4. The run-time computation of the maximum size of X ($2^{31} - 1 + 4 + 4$) thus raises CONSTRAINT_ERROR. Replacing NATURAL'LAST with NATURAL'LAST – 8 in the definition of STRING_SIZE will allow the maximum size to be computed; however, the compiler

will then attempt to allocate the maximum size ($2^{31} - 1$), and STORAGE_ERROR is raised.

One possible solution is to declare a subtype, STRING_SIZE, with a more realistic range, so that neither CONSTRAINT_ERROR nor STORAGE_ERROR will be raised. For example:

```
subtype STRING_SIZE is NATURAL range 0..256;
```

Another possibility is to declare the type as follows:

```
subtype V_STRING_SIZE is NATURAL range 0..256;
type V_STRING is
   record
      CURRENT_LAST: V_STRING_SIZE;
      S: STRING(1..V_STRING_SIZE'LAST);
   end record;

V: V_STRING;
```

This formulation is similar to a PL/I varying string. Assignments involve setting the component that indicates the current end of the string, and then using slice assignments to set the relevant portions of the text. For example, the following procedure appends VS2 to VS1:

```
procedure APPEND (VS1: in out V_STRING; VS2: in out V_STRING) is
begin
   VS1.S(VS1.CURRENT_LAST+1..VS1.CURRENT_LAST+VS2.CURRENT_LAST) :=
      VS2.S(1..VS2.CURRENT_LAST);
   VS1.CURRENT_LAST := VS1.CURRENT_LAST + VS2.CURRENT_LAST;
end;
```

A third possible solution is to use an intermediate access type. For example:

```
type ACCESS_STRING is access STRING;
X: ACCESS_STRING;
 . . .
X := new STRING'("This can be as long a string as you need!");
```

10.4 Assigning Array Values

When you assign array values, consider that Chapter 5 of the *VAX Ada Language Reference Manual* has some specific rules about assignments. In particular, note that bounds sliding does not occur in the following cases:

- During assignment of a record having array components
- During execution of a return statement

For example, consider the following procedure:

```
procedure SUBSTR is

   S1: constant STRING(1..10) := "1234567890";
   S2: STRING(1..5);
   type REC is record
      INT: INTEGER;
      STR: STRING(1..5);
   end record;
   R1: REC;

begin

   S2 := S1(6..10);          -- Assignment is ok.
   R1 := (555, S1(6..10)); -- Assignment unconditionally raises
                             -- CONSTRAINT_ERROR if executed.
   declare
      subtype S_1_TO_5 is STRING(1..5);
      function F return S_1_TO_5 is
      begin
         return S1(6..10); -- Assignment unconditionally raises
                           -- CONSTRAINT_ERROR if executed.
      end F;
   begin
      null;
   end;

end SUBSTR;
```

In this procedure, the assignment to S2 follows the rules for array assignments because the variable on the left side is an array variable (see Section 5.2.1 of the *VAX Ada Language Reference Manual*):

```
S2 := S1(6..10);
```

Thus, the expression S1(6..10) is implicitly converted to the subtype of the left side (STRING(1..5)); that is, bounds sliding occurs.

Consider the same procedure, rewritten with explicit subtypes to better show what is happening:

```
procedure SUBSTR is

   subtype S_1_TO_5 is STRING(1..5);
   subtype S_6_TO_10 is STRING(6..10);
   S1_LAST_PART: constant S_6_TO_10 :=
      (6 => '6', 7 => '7', 8 => '8', 9 => '9', 10 => '0');
   S2: S_1_TO_5;

begin

   S2 := S_1_TO_5(S1_LAST_PART); -- Array type conversion.

end SUBSTR;
```

In the assignment to S2 in this example, the bounds of S2 are 1..5, but the bounds of S1_LAST_PART are 6..10. The array type conversion converts the bounds of S1_LAST_PART to the bounds of S2. According to Chapter 5 of the *VAX Ada Language Reference Manual*, you could also write this assignment as follows, in which case the compiler would do the same conversion implicitly:

```
S2 := S1_LAST_PART
```

In contrast, the array assignment rules in Section 5.2.1 of the *VAX Ada Language Reference Manual* do not apply to the assignment to R1, because R1 is not an array variable:

```
R1 := (555, S1(6..10));
```

Here, the rules in Section 5.2 of the *VAX Ada Language Reference Manual* apply. According to these rules, the value of the expression (S1(6..10)) must be checked to see if it belongs to the subtype of the variable on the left side (STRING(1..5)), but no mention is made of implicit subtype conversions. Thus, this assignment raises the exception CONSTRAINT_ERROR because the bounds of the slice (6..10) do not match the bounds of STR (1..5).

Likewise, the rules for the return statement do not specify that an implicit subtype conversion should be done. Thus, the return statement also raises CONSTRAINT_ERROR:

```
return S1(6..10);
```

You can get the desired effect (bounds sliding) by using explicit array type conversions. The following example rewrites the procedure SUBSTR again, this time using type conversions in the return statement and the assignment to R1:

```
procedure SUBSTR is

   S1: STRING(1..10);
   subtype S5_SUBTYPE is STRING(1..5);
   type REC is record
      INT: INTEGER;
      STR: S5_SUBTYPE;
   end record;
   R1: REC;

   function F return S5_SUBTYPE is
   begin
      return S5_SUBTYPE(S1(6..10));     -- Type conversion.
   end;

begin

   R1 := (555, S5_SUBTYPE(S1(6..10))); -- Type conversion.
end SUBSTR;
```

10.5 Sharing Memory Between VAX CPUs

Example 10–2 shows how to write a VAX Ada program that shares memory between two or more VAX Ada programs running on one or more CPUs on a VMS system.

The program uses VMS global sections to share memory between processors. It does not use the lock manager for communicating between processes. The approach used in this example is recommended if there are fewer processes than CPUs. If there are more processes than CPUs, using the VMS lock manager may significantly improve performance.

Example 10–2: Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
--
--  First, declare a package that has a record type
--  (SHARED_OBJECTS_TYPE) for the data that is to be shared.
--  This record type should not have any task or access
--  components. It should have an INTERLOCK component.
--
--  You can modify this approach to use a SHARED_OBJECTS_TYPE
--  that could be specified in a pragma SHARED. You would use
--  the same technique to allocate the variable in shared memory.
--
with SYSTEM;
package SHARED_MEMORY_DATA_TYPES is

   type SHARED_OBJECTS_TYPE is
      record
         INTERLOCK       : SYSTEM.ALIGNED_WORD := (VALUE => 1);
         VALUE_AVAILABLE : BOOLEAN;
         VALUE           : INTEGER;
      end record;
```

(continued on next page)

Example 10–2 (Cont.): Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
   for SHARED_OBJECTS_TYPE use
      record
         INTERLOCK       at 0 range 0..15;
         VALUE_AVAILABLE at 4 range 0..7;
         VALUE           at 8 range 0..31;
      end record;

end SHARED_MEMORY_DATA_TYPES;

--  Next, write a procedure that can call the VMS SYS$CRMPSC system
--  service to create memory that is shared between two processes.
--  In this example, a groupwide section is either created (if it
--  did not already exist) or is mapped. The caller is returned two
--  pieces of information:
--
--     o  The address of the section (in this process's address
--        space; it may be at a different address in a different
--        process)
--
--     o  A boolean value indicating whether or not this was the
--        creating call to SYS$CRMPSC
--
with SYSTEM;
procedure CREATE_GLOBAL_SECTION(
   NAME            : STRING;
   SIZE            : NATURAL;
   SECTION_ADDRESS : out SYSTEM.ADDRESS;
   CREATED         : out BOOLEAN);

with STARLET, CONDITION_HANDLING, LIB;
procedure CREATE_GLOBAL_SECTION(
   NAME            : STRING;
   SIZE            : NATURAL;
   SECTION_ADDRESS : out SYSTEM.ADDRESS;
   CREATED         : out BOOLEAN) is

   STATUS : CONDITION_HANDLING.COND_VALUE_TYPE;

   INADR,
   RETADR : STARLET.ADDRESS_RANGE_TYPE :=
      (others => SYSTEM.ADDRESS_ZERO);

   FAILED_TO_CREATE_SECTION : exception;

   MATCH_COND_RESULT : SYSTEM.UNSIGNED_LONGWORD;
```

(continued on next page)

Example 10–2 (Cont.): Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
begin
   STARLET.CRMPSC (
      STATUS => STATUS,
      INADR  => INADR,
      RETADR => RETADR,
      FLAGS  => SYSTEM.UNSIGNED_LONGWORD(STARLET.SEC_M_GBL +
                                         STARLET.SEC_M_DZRO +
                                         STARLET.SEC_M_EXPREG +
                                         STARLET.SEC_M_WRT +
                                         STARLET.SEC_M_PAGFIL),
      GSDNAM => NAME,
      PAGCNT => SYSTEM.UNSIGNED_LONGWORD(((SIZE+7)/8+511)/512),
                                           -- W    G    O    S
                                           --DEWR DEWR DEWR DEWR
      PROT => STARLET.FILE_PROTECTION_TYPE'(2#0000_0000_1011_0000#));

   if not CONDITION_HANDLING.SUCCESS(STATUS) then
      raise FAILED_TO_CREATE_SECTION;
   end if;

   LIB.MATCH_COND(MATCH_COND_RESULT, STATUS, STARLET.SS_CREATED);

   SECTION_ADDRESS := RETADR(0);
   CREATED         := SYSTEM."="(MATCH_COND_RESULT, 1);

end CREATE_GLOBAL_SECTION;

-- Now, write a function that uses the procedure
-- CREATE_GLOBAL_SECTION to place the particular set of objects
-- to be shared into shared memory.
--
-- The call that creates the shared memory initializes the objects.
--
-- This function sets up a race condition: other callers may arrive
-- after the section has been created, but before it is initialized.
-- There are a number of ways to handle this situation.  The approach
-- in this example is for the user to start running the programs that
-- use the shared variable only after the first program has created
-- and initialized it.
--
with SYSTEM;
function CREATE_MY_SHARED_OBJECTS return SYSTEM.ADDRESS;

with SHARED_MEMORY_DATA_TYPES, CREATE_GLOBAL_SECTION,
     SYSTEM, TEXT_IO;
pragma ELABORATE(CREATE_GLOBAL_SECTION);
```

(continued on next page)

Example 10–2 (Cont.): Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
function CREATE_MY_SHARED_OBJECTS return SYSTEM.ADDRESS is

   SECTION_ADDRESS : SYSTEM.ADDRESS;
   CREATED         : BOOLEAN;

begin

   CREATE_GLOBAL_SECTION(
      "SHARED_MEMORY",
      SHARED_MEMORY_DATA_TYPES.SHARED_OBJECTS_TYPE'MACHINE_SIZE,
      SECTION_ADDRESS,
      CREATED);

   if CREATED then

      -- Cause the shared variable to be initialized; this should
      -- happen only once.
      --
      declare

         SHARED_OBJECTS :
            SHARED_MEMORY_DATA_TYPES.SHARED_OBJECTS_TYPE;
               for SHARED_OBJECTS use at SECTION_ADDRESS;
               pragma VOLATILE(SHARED_OBJECTS);

      begin
         null;
      end;

      TEXT_IO.PUT_LINE("Section is created and initialized. " &
                       "Start other programs now.");

   end if;
   return SECTION_ADDRESS;

end CREATE_MY_SHARED_OBJECTS;

-- Now, use a piece of clever Ada code to make a widely visible
-- object appear in the area of shared memory.
--
with SYSTEM, UNCHECKED_CONVERSION;
with SHARED_MEMORY_DATA_TYPES, CREATE_MY_SHARED_OBJECTS;
pragma ELABORATE(CREATE_MY_SHARED_OBJECTS);

package SHARED_MEMORY is

   type A is access SHARED_MEMORY_DATA_TYPES.SHARED_OBJECTS_TYPE;
```

(continued on next page)

Example 10–2 (Cont.): Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
   function TO_A is new UNCHECKED_CONVERSION(SYSTEM.ADDRESS, A);

   -- Here is the key to this code: by renaming a '.all' construct,
   -- the code makes an object visible, but ensures that it will not
   -- have initialization problems.
   --
   SHARED_OBJECTS : SHARED_MEMORY_DATA_TYPES.SHARED_OBJECTS_TYPE
                    renames TO_A(CREATE_MY_SHARED_OBJECTS).all;

   -- Provide simple names for the components.
   --
   VALUE           : INTEGER renames SHARED_OBJECTS.VALUE;
   VALUE_AVAILABLE : BOOLEAN renames SHARED_OBJECTS.VALUE_AVAILABLE;

   -- Provide an interlock.  The use of an interlocked instruction
   -- (SYSTEM.ADD_INTERLOCKED) within these statements causes all
   -- memory modifications to be flushed through the multiCPU caches
   -- and become visible to all participating processes.
   --
   -- The use of spinning and a delay statement means that things
   -- do not stall for very long, and other tasks within the program
   -- can continue while one stalls waiting for the lock.  The
   -- locking shown here does not support multiple simultaneous
   -- readers.  Extending to that case is straightforward, and does
   -- not affect the rest of this example.
   --
   procedure ACQUIRE;
   procedure RELEASE;

end SHARED_MEMORY;

with SYSTEM;
package body SHARED_MEMORY is

   INTERLOCK : SYSTEM.ALIGNED_WORD renames SHARED_OBJECTS.INTERLOCK;

   MAX_SPIN_COUNT : constant POSITIVE := 10;

   procedure ACQUIRE is
      SIGN : INTEGER;
      SPIN_COUNT : INTEGER := 0;
   begin
      loop
         SYSTEM.ADD_INTERLOCKED(-1, INTERLOCK, SIGN);
         exit when SIGN = 0;
```

(continued on next page)

Example 10–2 (Cont.): Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
         SYSTEM.ADD_INTERLOCKED( 1, INTERLOCK, SIGN);
         SPIN_COUNT := SPIN_COUNT + 1;
         if SPIN_COUNT > MAX_SPIN_COUNT then
            delay DURATION'SMALL;
         end if;
      end loop;
   end;

   procedure RELEASE is
      SIGN : INTEGER;
   begin
      SYSTEM.ADD_INTERLOCKED( 1, INTERLOCK, SIGN);
   end;

end SHARED_MEMORY;

-- Here is a main program that is going to write values into the
-- shared memory. It waits until each previous value is read by
-- a reader before writing the next value.
--
with SHARED_MEMORY, TEXT_IO;
use  SHARED_MEMORY, TEXT_IO;
procedure Z_SHARE_MEMORY_WRITER is
begin
   for I in NATURAL loop
      loop
         ACQUIRE;
         if not VALUE_AVAILABLE then
            VALUE_AVAILABLE := TRUE;
            VALUE := I;
            PUT_LINE("Writing VALUE =" & INTEGER'IMAGE(VALUE));
            RELEASE;
            exit;
         end if;
         RELEASE;
      end loop;
   end loop;
end Z_SHARE_MEMORY_WRITER;
```

(continued on next page)

Example 10–2 (Cont.): Sharing Memory Between Two or More Programs Running on One or More VAX CPUs

```
-- Here are two readers (each a main program in itself).
--
with SHARED_MEMORY, TEXT_IO;
use  SHARED_MEMORY, TEXT_IO;
procedure Z_SHARE_MEMORY_READER1 is
begin
   loop
      ACQUIRE;

      if VALUE_AVAILABLE then
         VALUE_AVAILABLE := FALSE;
         PUT_LINE("READER1 VALUE =" & INTEGER'IMAGE(VALUE));
      end if;
      RELEASE;
   end loop;
end Z_SHARE_MEMORY_READER1;

with SHARED_MEMORY, TEXT_IO;
use  SHARED_MEMORY, TEXT_IO;
procedure Z_SHARE_MEMORY_READER2 is
begin
   loop
      ACQUIRE;
      if VALUE_AVAILABLE then
         VALUE_AVAILABLE := FALSE;
         PUT_LINE("READER2 VALUE =" & INTEGER'IMAGE(VALUE));
      end if;
      RELEASE;
   end loop;
end Z_SHARE_MEMORY_READER2;
```

Appendix A

VAX Ada Predefined Instantiations

For convenience, and for the purpose of saving compilation time and object code space, VAX Ada predefines the instantiations of some commonly used generic packages:

Unit Name	Instantiation of	For Type
INTEGER_TEXT_IO	TEXT_IO.INTEGER_IO	INTEGER
SHORT_INTEGER_TEXT_IO	TEXT_IO.INTEGER_IO	SHORT_INTEGER
SHORT_SHORT_INTEGER_TEXT_IO	TEXT_IO.INTEGER_IO	SHORT_SHORT_INTEGER
FLOAT_TEXT_IO	TEXT_IO.FLOAT_IO	FLOAT
LONG_FLOAT_TEXT_IO	TEXT_IO.FLOAT_IO	LONG_FLOAT
LONG_LONG_FLOAT_TEXT_IO	TEXT_IO.FLOAT_IO	LONG_LONG_FLOAT
FLOAT_MATH_LIB	MATH_LIB	FLOAT
LONG_FLOAT_MATH_LIB	MATH_LIB	LONG_FLOAT
LONG_LONG_FLOAT_MATH_LIB	MATH_LIB	LONG_LONG_FLOAT

The representation used for the type LONG_FLOAT in these packages is G_floating. Thus, to use LONG_FLOAT_TEXT_IO and LONG_FLOAT_MATH_LIB, you must be sure that the G_floating representation for LONG_FLOAT is in effect for any compilations involving these packages. Note the following information:

- G_floating is the default whenever you create or reinitialize a program library or sublibrary.

- You can set the representation for LONG_FLOAT either with the VAX Ada pragma LONG_FLOAT or with a number of program library manager commands (ACS SET PRAGMA, ACS CREATE LIBRARY/LONG_FLOAT, and ACS CREATE SUBLIBRARY/LONG_FLOAT).
- You can determine whether or not G_floating is in effect for a program library or sublibrary by first making the library the current library (use the ACS SET LIBRARY command) and then entering an ACS SHOW PROGRAM or ACS DIRECTORY command.

If you change the representation of the type LONG_FLOAT to D_floating for your current program library, you will need to recompile the instantiations LONG_FLOAT_TEXT_IO and LONG_FLOAT_MATH_LIB in the context of the D_floating representation in order to use them. To do this, extract the source of these packages into the current program library, and then compile them using the DCL ADA command (or ACS LOAD and COMPILE commands). For example:

```
$ ACS SET PRAGMA/LONG_FLOAT=D_FLOAT
$ ACS EXTRACT SOURCE LONG_FLOAT_TEXT_IO
    .
    .
    .
$ ADA LONG_FLOAT_TEXT_IO
```

Any change in the setting of the representation for the type LONG_FLOAT implies a recompilation of the predefined STANDARD environment.

See Section 3.5.7a of the *VAX Ada Language Reference Manual* for more information on the pragma LONG_FLOAT. See *Developing Ada Programs on VMS Systems* for more information on ACS commands, compiling Ada programs, and the implied recompilation of the predefined STANDARD environment.

See Chapter 3 of this manual for information on using the predefined instantiations for TEXT_IO.INTEGER_IO and TEXT_IO.FLOAT_IO. See Chapter 6 of this manual for information on using the predefined instantiations of MATH_LIB.

Appendix B

VAX Ada Packages

VAX Ada provides the packages listed in Table B–1. As noted in Table B–1, this appendix and the *VAX Ada Language Reference Manual* provide specifications or parts of the specifications for some of these packages.

You can obtain the complete specifications and, in some cases, the bodies for any of these packages by using the ACS EXTRACT SOURCE command. For example, the following command causes the specifications of the packages STARLET, STANDARD, and TEXT_IO, to be placed in your current default directory:

```
$ ACS EXTRACT SOURCE STARLET, $STANDARD, TEXT_IO
```

You must have defined a current program library to execute this command. The current program library can be either the library ADA$PREDEFINED or a library into which the predefined units from ADA$PREDEFINED have been entered. See *Developing Ada Programs on VMS Systems* or type HELP ACS EXTRACT SOURCE at the VMS prompt for more information.

Table B–1: VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
ASSERT	Instantiation of the package ASSERT_GENERIC. Assumes all of the defaults in the package ASSERT_GENERIC, including the use of the procedure TEXT_IO.PUT_LINE to report failures.	*VAX Ada Run-Time Reference Manual*, Appendix B
ASSERT_EXCEPTIONS	Declares all exceptions that can be raised by instantiations of the package ASSERT_GENERIC.	*VAX Ada Run-Time Reference Manual*, Appendix B
ASSERT_GENERIC	Provides types and operations that allow you to insert and enable code-checking assertions in your Ada source code. Depends on the packages ASSERT_EXCEPTIONS and TEXT_IO.	*VAX Ada Run-Time Reference Manual*, Appendix B
AUX_IO_EXCEPTIONS	Defines the exceptions needed by the VAX Ada relative and indexed input-output packages.	*VAX Ada Language Reference Manual*, Chapter 14
CALENDAR	Provides time-related types and operations. Depends on the package SYSTEM.	*VAX Ada Language Reference Manual*, Chapter 9

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
CDD_TYPES	Provides Ada equivalents for VAX CDD data types; additional equivalents are in the packages STANDARD and SYSTEM. Depends on the package SYSTEM.	*VAX Ada Run-Time Reference Manual*, Appendix B
CLI	Provides types and operations for calling VMS Command Language Utility routines. Depends on the packages SYSTEM and CONDITION_HANDLING.	—
CONDITION_HANDLING	Provides types and operations needed to evaluate the condition values returned by system routines. Depends on the package SYSTEM.	*VAX Ada Run-Time Reference Manual*, Appendix B
CONTROL_C_INTERCEPTION	Establishes a VAX Ada CTRL/C handler when it is elaborated.	*VAX Ada Run-Time Reference Manual*, Appendix B
DIRECT_IO	Provides types and operations for working with direct files of uniform-type elements. Depends on the package IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14
DIRECT_MIXED_IO	Provides types and operations for working with direct files of mixed-type elements. Depends on the package IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
DTK	Provides types and operations for calling the VMS Run-Time Library DTK$ routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
INDEXED_IO	Provides types and operations for working with indexed files of uniform-type elements. Depends on the packages IO_EXCEPTIONS and AUX_IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14
INDEXED_MIXED_IO	Provides types and operations for working with indexed files of mixed-type elements. Depends on the packages IO_EXCEPTIONS and AUX_IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14
IO_EXCEPTIONS	Defines exceptions needed by all of the input-output packages.	*VAX Ada Language Reference Manual*, Chapter 14
LBR	Provides types and operations for calling the VMS Librarian Utility routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
LIB	Provides types and operations for calling the VMS Run-Time Library LIB$ routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
MATH_LIB	Provides a set of operations and exceptions that correspond to some of the VMS Run-Time Library Mathematical Library routines and conditions.	*VAX Ada Run-Time Reference Manual*, Appendix B
MTH	Provides types and operations for calling the VMS Run-Time Library MTH$ routines. Depends on the package SYSTEM.	—
NCS	Provides types and operations for calling the National Character Set Utility routines. Depends on the packages SYSTEM and CONDITION_HANDLING.	—
OTS	Provides types and operations for calling the VMS Run-Time Library OTS$ routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
PPL	Provides types and operations for calling the VMS Run-Time Library PPL$ routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
RELATIVE_IO	Provides types and operations for working with relative files of uniform-type elements. Depends on the packages IO_EXCEPTIONS and AUX_IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14
RELATIVE_MIXED_IO	Provides types and operations for working with relative files of mixed-type elements. Depends on the packages IO_EXCEPTIONS and AUX_IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14
RMS_ASYNCH_OPERATIONS	Provides supporting operations for the package TASKING_SERVICES. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
SEQUENTIAL_IO	Provides types and operations for working with sequential files of uniform-type elements. Depends on the package IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14
SEQUENTIAL_MIXED_IO	Provides types and operations for working with sequential files of mixed-type elements. Depends on the package IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
SMG	Provides types and operations for calling the VMS Run-Time Library SMG$ routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
SOR	Provides types and operations for calling the Sort/Merge Utility routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
STANDARD	Provides all the predefined types, operations, and exceptions defined by the language, as well as the additional VAX Ada types SHORT_INTEGER, SHORT_SHORT_INTEGER, LONG_FLOAT, and LONG_LONG_FLOAT.	*VAX Ada Language Reference Manual*, Chapter 8 and Annex C
STARLET	Provides the types, operations, constants, and so on that you need to call VMS system service and RMS routines. Depends on the packages SYSTEM and CONDITION_HANDLING.	*VAX Ada Run-Time Reference Manual*, Appendix B: type declarations only

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
STR	Provides types and operations for calling the VMS Run-Time Library STR$ routines. Depends on the packages SYSTEM, STARLET, and CONDITION_HANDLING.	—
SYSTEM	Provides implementation-defined types, operations, constants, and named numbers, some of which are required by the language standard, and some of which are provided by VAX Ada.	*VAX Ada Language Reference Manual*, Chapter 13 and Appendix F
SYSTEM_RUNTIME_TUNING	Provides operations for changing system parameters that are normally controlled by the VAX Ada run-time library.	*VAX Ada Run-Time Reference Manual*, Appendix B
TASKING_SERVICES	Provides task-synchronous, process-asynchronous forms of the VMS system services SYS$BRKTHRU, SYS$ENQ, SYS$GETDVI, SYS$GETJPI, SYS$GETLKI, SYS$GETQUI, SYS$GETSYI, SYS$QIO, SYS$SNDJBC, SYS$UPDSEC, and any VMS RMS record or block operation. Depends on the packages SYSTEM, STARLET, CONDITION_HANDLING, and RMS_ASYNCH_OPERATIONS.	*VAX Ada Run-Time Reference Manual*, Appendix B

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table B–1 (Cont.): VAX Ada Predefined Packages

Package Name	Description	Location of Specification in the VAX Ada Documentation Set[1]
TEXT_IO	Provides types and operations for working with text files. Depends on the package IO_EXCEPTIONS.	*VAX Ada Language Reference Manual*, Chapter 14

[1]All of the package specifications are also available from the VAX Ada library of predefined units, ADA$PREDEFINED.

B.1 Packages ASSERT, ASSERT_EXCEPTIONS, and ASSERT_GENERIC

```
with ASSERT_GENERIC;
package ASSERT is new ASSERT_GENERIC;

    -- This package is a predefined instantiation of the package
    -- ASSERT_GENERIC.  It assumes all of the default values defined in
    -- the package ASSERT_GENERIC, including the use of the procedure
    -- TEXT_IO.PUT_LINE to report assertion warnings and failures.

package ASSERT_EXCEPTIONS is

    -- This package declares the exceptions that can be raised by any
    -- instantiations of the package ASSERT_GENERIC.
    --
    ASSERT_ERROR : exception;
end;

with ASSERT_EXCEPTIONS;
with TEXT_IO;
generic

    -- This package provides types and operations that allow you to put
    -- debuggging checks, or assertions, in your Ada programs. The
    -- package defines two kinds of assertions: warnings and failures.
    --
    -- Features:
    --
    -- o A number of generic parameters with default values are provided
    --   that allow you to control the effect of assertions on the
    --   generated code, the action routines to be called when assertion
    --   is true, and so on.
    --
    -- o The default action routine for announcing assertion warnings and
    --   failures is TEXT_IO.PUT_LINE.
    --
    -- o By instantiating the generic package with the CHECK_WARNINGS and
    --   CHECK_FAILURES parameters set to FALSE, you can eliminate code
    --   generation for your assertions without having to change all of the
    --   routines that call into the assertion package. Thus, you can leave
    --   out assertion checks for your production programs.
    --
    -- o No code is generated for assertions that are
    --   statically true.
    --
    --
    -- EXAMPLE: Using the preinstantiated package ASSERT.
    --
    --     with ASSERT;
    --     ...
    --     ASSERT.WARN(expression, "failure description");
```

```
--     ASSERT.FAIL(expression, "failure description");
--
-- EXAMPLE: Creating your own assertion package that will
--          hide the preinstantiated package and disable the
--          code for all checks.
--
--    with ASSERT_GENERIC;
--    package ASSERT is new ASSERT_GENERIC( FALSE, FALSE );
--
-- EXAMPLE: Creating your own assertion package that will
--          hide the preinstantiated package and that will
--          use a special action routine.
--
--    with ASSERT_GENERIC;
--    procedure MAIN_PROGRAM is
--       ...
--       package ASSERT is
--          new ASSERT_GENERIC(FAIL_ACTION_ROUTINE => SPECIAL_REPORT);
--
--
--
-- Determine if the failure (or warning) assertion checks will
-- actually be inserted into the generated code of the caller of the
-- FAIL (or WARN) routine. If these values are false, and inline
-- optimization is enabled, there should be little or no run-time
-- impact on any caller of the assertion routine.
--
CHECK_WARNINGS : in BOOLEAN := TRUE;
CHECK_FAILURES : in BOOLEAN := TRUE;

-- Action routine to be called when a warning assertion is false.
-- The value of the ITEM parameter is concatenated with two other
-- strings when it is output: the WARN_PREFIX_TEXT string and the
-- DESCRIPTION string (which is passed as a parameter to the WARN
-- procedure).  If the value of WARN_LIMIT (see private declaration)
-- is greater than 0, only that number of calls is made to the action
-- routine, and the last call is followed by the raising of exception
-- ASSERT_ERROR.
--
-- NOTE: Subsequent assertion warnings just raise the exception.
--
with procedure WARN_ACTION_ROUTINE(ITEM : STRING) is TEXT_IO.PUT_LINE;
```

```
    -- Action routine to be called when a failure assertion is false.
    -- The value of the ITEM parameter is concatenated with two other
    -- strings when it is ouput: the FAIL_PREFIX_TEXT string and the
    -- DESCRIPTION string (which is passed as a parameter to the FAIL
    -- procedure).  If the value of FAIL_LIMIT (see private declaration)
    -- is greater than 0, only that number of calls is made to the action
    -- routine, and the last call is followed by the raising of exception
    -- ASSERT_ERROR.
    --
    -- NOTE: Subsequent assertion failures just raise the exception.
    -- The default value of FAIL_LIMIT is 1.
    --
    with procedure FAIL_ACTION_ROUTINE(ITEM : STRING) is TEXT_IO.PUT_LINE;

    -- The maximum number of times that the FAIL_ACTION_ROUTINE (or
    -- WARN_ACTION_ROUTINE) will be called.  If 0, then there is no limit
    -- to how many times the action routine will be called.
    --
    -- NOTE: The values for WARN_LIMIT_INIT and FAIL_LIMIT_INIT
    -- initialize the tailoring parameters WARN_LIMIT and FAIL_LIMIT (see
    -- private declarations).
    --
    WARN_LIMIT_INIT : in NATURAL := 50;
    FAIL_LIMIT_INIT : in NATURAL := 1;

    -- When set to TRUE, inhibits the raising of the exception when the
    -- value of WARN_LIMIT or FAIL_LIMIT (see private declaration) is
    -- reached or exceeded.
    --
    -- NOTE: The value for INHIBIT_EXCEPTION_INIT initializes the
    -- tailoring parameter INHIBIT_EXCEPTION (see private declaration).
    --
    INHIBIT_EXCEPTION_INIT : in BOOLEAN := FALSE;

    -- Strings used to prefix action routine messages.
    --
    FAIL_PREFIX_TEXT : in STRING := "**** Assertion failure. ";
    WARN_PREFIX_TEXT : in STRING := "**** Assertion warning. ";


package ASSERT_GENERIC is

    -- The exception raised on assertion failures (unless inhibited by
    -- INHIBIT_EXCEPTION).
    --
    ASSERT_ERROR : exception renames ASSERT_EXCEPTIONS.ASSERT_ERROR;
```

```
    -- If the boolean condition is FALSE, the WARN_ACTION_ROUTINE is
    -- called and passed the DESCRIPTION string.  If the value of
    -- WARN_LIMIT has been reached, the action routine is called and
    -- the exception ASSERT_ERROR is raised; any subsequent assertion
    -- warnings result only in the exception being raised.  If the value
    -- of CHECK_WARNINGS is FALSE, little or no code is generated in the
    -- caller of this routine, provided inline optimizations are not
    -- inhibited.
    --
    procedure WARN(ASSERTION   : BOOLEAN := FALSE;
                   DESCRIPTION : STRING  := "");
    pragma INLINE(WARN);

    -- If the boolean condition is FALSE, the FAIL_ACTION_ROUTINE is
    -- called and passed the DESCRIPTION string.  If the value of
    -- FAIL_LIMIT has been reached, the action routine is called and the
    -- exception ASSERT_ERROR is raised; any subsequent assertion
    -- warnings result only in the exception being raised. If the value
    -- of CHECK_FAILURES is FALSE, little or no code is generated in the
    -- caller of this routine, provided inline optimizations are not
    -- inhibited.
    --
    -- NOTE: You cannot rely on exceptions terminating the program
    -- because your program might absorb the exception and continue
    -- execution.  The following functions are provided so that your
    -- program can check if any warnings or failures occurred:
    --
    procedure FAIL(ASSERTION   : BOOLEAN := FALSE;
                   DESCRIPTION : STRING  := "");
    pragma INLINE(FAIL);

    -- A means of checking if an assertion warning or failure has
    -- occurred in the program.
    --
    function HAD_ASSERT_WARNING return BOOLEAN;

    -- A means of checking if an assertion failure has occurred in the
    -- program.
    --
    function HAD_ASSERT_FAILURE return BOOLEAN;

private

    -- Tailoring Parameters.
    --
    -- The following parameters are initialized when the generic package is
    -- instantiated, but you can change them when you execute the program
    -- under the control of the VMS Debugger.  By changing their values
    -- with debugger commands, you can modify the treatment of assertion
    -- failures while you are debugging.  The initial values are in the
    -- private part so that they cannot be changed by the program as it
    -- executes without the debugger.
```

```
    -- When its value is TRUE, it inhibits the raising of assertion
    -- exceptions.
    --
    INHIBIT_EXCEPTION  : BOOLEAN := INHIBIT_EXCEPTION_INIT;

    -- These values determine the maximum number of times that the
    -- respective action routine will be called.  If they are set to 0,
    -- then the action routine is called on every failure (or warning).
    --
    WARN_LIMIT   : NATURAL := WARN_LIMIT_INIT;
    FAIL_LIMIT   : NATURAL := FAIL_LIMIT_INIT;

end ASSERT_GENERIC;

pragma INLINE_GENERIC (ASSERT_GENERIC);
```

B.2 Package CDD_TYPES

```
with SYSTEM; use SYSTEM;
package CDD_TYPES is

    -- This package defines types that are needed by the CDD-to-Ada
    -- translator (ADA$FROM_CDD).

    -- Special names to use for unsupported data types.
    --
    subtype UNSUPPORTED_TYPE1 is BIT_ARRAY;
    subtype UNSUPPORTED_TYPE2 is UNSIGNED_BYTE_ARRAY;

    -- Declare an octaword type.
    --
    subtype OCTAWORD_TYPE is UNSIGNED_LONGWORD_ARRAY(0 .. 3);

    -- Declare a date/time.
    --
    subtype DATE_TIME_TYPE is UNSIGNED_QUADWORD;

    -- For completeness, show BIT_ARRAY (it is defined in
    -- the package SYSTEM):
    --
    --     type BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
    --
    type BIT_ARRAY_2D is
        array (INTEGER range <>,
               INTEGER range <>) of BOOLEAN;
    type BIT_ARRAY_3D is
        array (INTEGER range <>,
               INTEGER range <>,
               INTEGER range <>) of BOOLEAN;
    pragma PACK(BIT_ARRAY_2D);
    pragma PACK(BIT_ARRAY_3D);
```

```
-- For completeness, show STRING (it is defined in
-- the package STANDARD):
--
--     type STRING is array (POSITIVE range <>) of CHARACTER;
--
type STRING_2D is
    array (INTEGER range <>,
           INTEGER range <>) of CHARACTER;
type STRING_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of CHARACTER;

-- For completeness, show UNSIGNED_BYTE_ARRAY (it is defined in
-- the package SYSTEM):
--
--     type UNSIGNED_BYTE_ARRAY is
--          array (INTEGER range <>) of UNSIGNED_BYTE;
--
type UNSIGNED_BYTE_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of UNSIGNED_BYTE;
type UNSIGNED_BYTE_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of UNSIGNED_BYTE;

-- For completeness, show UNSIGNED_WORD_ARRAY (it is defined in
-- the package SYSTEM):
--
--     type UNSIGNED_WORD_ARRAY is
--          array (INTEGER range <>) of UNSIGNED_WORD;
--
type UNSIGNED_WORD_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of UNSIGNED_WORD;
type UNSIGNED_WORD_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of UNSIGNED_WORD;

-- For completeness, show UNSIGNED_LONGWORD_ARRAY (it is defined in
-- the package SYSTEM):
--
--     type UNSIGNED_LONGWORD_ARRAY is
--          array (INTEGER range <>) of UNSIGNED_LONGWORD;
--
type UNSIGNED_LONGWORD_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of UNSIGNED_LONGWORD;
type UNSIGNED_LONGWORD_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of UNSIGNED_LONGWORD;
```

```
-- For completeness, show UNSIGNED_QUADWORD_ARRAY (it is defined in
-- the package SYSTEM):
--
--     type UNSIGNED_QUADWORD_ARRAY is
--          array (INTEGER range <>) of UNSIGNED_QUADWORD;
--
type UNSIGNED_QUADWORD_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of UNSIGNED_QUADWORD;
type UNSIGNED_QUADWORD_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of UNSIGNED_QUADWORD;

type SHORT_SHORT_INTEGER_ARRAY is
    array (INTEGER range <>) of SHORT_SHORT_INTEGER;
type SHORT_SHORT_INTEGER_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of SHORT_SHORT_INTEGER;
type SHORT_SHORT_INTEGER_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of SHORT_SHORT_INTEGER;

type SHORT_INTEGER_ARRAY is array (INTEGER range <>) of SHORT_INTEGER;
type SHORT_INTEGER_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of SHORT_INTEGER;
type SHORT_INTEGER_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of SHORT_INTEGER;

type INTEGER_ARRAY is array (INTEGER range <>) of INTEGER;
type INTEGER_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of INTEGER;
type INTEGER_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of INTEGER;

type FLOAT_ARRAY is array (INTEGER range <>) of FLOAT;
type FLOAT_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of FLOAT;
type FLOAT_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of FLOAT;
```

```
type D_FLOAT_ARRAY is array (INTEGER range <>) of D_FLOAT;
type D_FLOAT_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of D_FLOAT;
type D_FLOAT_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of D_FLOAT;

type LONG_LONG_FLOAT_ARRAY is
    array (INTEGER range <>) of LONG_LONG_FLOAT;
type LONG_LONG_FLOAT_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of LONG_LONG_FLOAT;
type LONG_LONG_FLOAT_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of LONG_LONG_FLOAT;

type G_FLOAT_ARRAY is array (INTEGER range <>) of G_FLOAT;
type G_FLOAT_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of G_FLOAT;
type G_FLOAT_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of G_FLOAT;

type OCTAWORD_ARRAY is array (INTEGER range <>) of OCTAWORD_TYPE;
type OCTAWORD_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of OCTAWORD_TYPE;
type OCTAWORD_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of OCTAWORD_TYPE;

type DATE_TIME_ARRAY is array (INTEGER range <>) of DATE_TIME_TYPE;
type DATE_TIME_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of DATE_TIME_TYPE;
type DATE_TIME_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of DATE_TIME_TYPE;

type ADDRESS_ARRAY is array (INTEGER range <>) of ADDRESS;
type ADDRESS_ARRAY_2D is
    array (INTEGER range <>,
           INTEGER range <>) of ADDRESS;
type ADDRESS_ARRAY_3D is
    array (INTEGER range <>,
           INTEGER range <>,
           INTEGER range <>) of ADDRESS;
```

```
    -- Define static subtypes of type INTEGER.
    --
    subtype SIGNED_1  is INTEGER range -(2** 0) ..  2** 0-1;
    subtype SIGNED_2  is INTEGER range -(2** 1) ..  2** 1-1;
    subtype SIGNED_3  is INTEGER range -(2** 2) ..  2** 2-1;
    subtype SIGNED_4  is INTEGER range -(2** 3) ..  2** 3-1;
    subtype SIGNED_5  is INTEGER range -(2** 4) ..  2** 4-1;
    subtype SIGNED_6  is INTEGER range -(2** 5) ..  2** 5-1;
    subtype SIGNED_7  is INTEGER range -(2** 6) ..  2** 6-1;
    subtype SIGNED_8  is INTEGER range -(2** 7) ..  2** 7-1;
    subtype SIGNED_9  is INTEGER range -(2** 8) ..  2** 8-1;

    subtype SIGNED_10 is INTEGER range -(2** 9) ..  2** 9-1;
    subtype SIGNED_11 is INTEGER range -(2**10) ..  2**10-1;
    subtype SIGNED_12 is INTEGER range -(2**11) ..  2**11-1;
    subtype SIGNED_13 is INTEGER range -(2**12) ..  2**12-1;
    subtype SIGNED_14 is INTEGER range -(2**13) ..  2**13-1;
    subtype SIGNED_15 is INTEGER range -(2**14) ..  2**14-1;
    subtype SIGNED_16 is INTEGER range -(2**15) ..  2**15-1;
    subtype SIGNED_17 is INTEGER range -(2**16) ..  2**16-1;
    subtype SIGNED_18 is INTEGER range -(2**17) ..  2**17-1;
    subtype SIGNED_19 is INTEGER range -(2**18) ..  2**18-1;

    subtype SIGNED_20 is INTEGER range -(2**19) ..  2**19-1;
    subtype SIGNED_21 is INTEGER range -(2**20) ..  2**20-1;
    subtype SIGNED_22 is INTEGER range -(2**21) ..  2**21-1;
    subtype SIGNED_23 is INTEGER range -(2**22) ..  2**22-1;
    subtype SIGNED_24 is INTEGER range -(2**23) ..  2**23-1;
    subtype SIGNED_25 is INTEGER range -(2**24) ..  2**24-1;
    subtype SIGNED_26 is INTEGER range -(2**25) ..  2**25-1;
    subtype SIGNED_27 is INTEGER range -(2**26) ..  2**26-1;
    subtype SIGNED_28 is INTEGER range -(2**27) ..  2**27-1;
    subtype SIGNED_29 is INTEGER range -(2**28) ..  2**28-1;

    subtype SIGNED_30 is INTEGER range -(2**29) ..  2**29-1;
    subtype SIGNED_31 is INTEGER range -(2**30) ..  2**30-1;

end;
```

B.3 Package CONDITION_HANDLING

```
with SYSTEM; use SYSTEM;
package CONDITION_HANDLING is

    -- This package defines the following:
    --
    --   1. The VMS condition value type
    --
    --   2. Functions for accessing the components of a
    --      condition value
    --
    --   3. Interfaces to the VMS Run-Time Library routines
    --      LIB$SIGNAL, LIB$STOP, and LIB$MATCH_COND
    --

    -- VMS system services return values of type COND_VALUE_TYPE.
    --
    subtype COND_VALUE_TYPE is SYSTEM.UNSIGNED_LONGWORD;

    -- VMS severity codes are defined in the package STARLET as:
    --
    --    STS_K_WARNING : constant := 0;
    --    STS_K_SUCCESS : constant := 1;
    --    STS_K_ERROR   : constant := 2;
    --    STS_K_INFO    : constant := 3;
    --    STS_K_SEVERE  : constant := 4;
```

```
-- You can obtain components of a condition value with
-- the following functions.  See Appendix C of the
-- VAX Architecture Handbook or the section on condition
-- handling in the Introduction to VMS System Services.
-- The partial representation specification following
-- each description indicates the portion of the condition
-- value returned by each of the functions.
--
--     SEVERITY
--          Severity code:  STS_K__WARNING, STS_K__SUCCESS,
--          STS_K__ERROR, STS_K__INFO, or STS_K__SEVERE.
--
--              for SEVERITY        at 0 range 0 .. 2;
--
--     SUCCESS
--          Is TRUE if low bit of severity is on.  This occurs
--          for a severity of STS_K__SUCCESS or STS_K__INFO.
--
--              for SUCCESS         at 0 range 0 .. 0;
--
--     COND_ID
--          Identifies the conditions uniquely on a system-wide basis.
--
--              for COND_ID         at 0 range 3 .. 27;
--
--     FAC_NO
--          Identifies the software component generating the condition
--          value.
--
--              for FAC_NO          at 0 range 16 .. 27;
--
--     CUST_DEF
--          Is TRUE for customer facilities and FALSE for Digital
--          facilities.
--
--              for CUST_DEF        at 0 range 27 .. 27;
--
--     MSG_NO
--          A status identification; that is, a description of the
--          hardware exception that occurred or a software-defined
--          value.
--
--              for MSG_NO          at 0 range 3 .. 15;
--
--     FAC_SP
--          Is TRUE for message numbers that are specific to a
--          single facility and FALSE for system-wide message numbers.
--
--              for FAC_SP          at 0 range 15 .. 15;
--
--     CODE
--          Message number (without facility specific flag).
--
```

```
--              for CODE            at 0 range 3 .. 14;
--
--     INHIB_MSG
--          Is TRUE if the message should be inhibited on image exit
--          and FALSE otherwise.
--
--              for INHIB_MSG       at 0 range 27 .. 27;
--
function SEVERITY  (STATUS : COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
function SUCCESS   (STATUS : COND_VALUE_TYPE)
    return BOOLEAN;
function COND_ID   (STATUS : COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
function FAC_NO    (STATUS : COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
function CUST_DEF  (STATUS : COND_VALUE_TYPE)
    return BOOLEAN;
function MSG_NO    (STATUS : COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
function FAC_SP    (STATUS : COND_VALUE_TYPE)
    return BOOLEAN;
function CODE      (STATUS : COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
function INHIB_MSG (STATUS : COND_VALUE_TYPE)
    return BOOLEAN;

-- Some system services use status blocks (for example, an
-- input-output status block) that contain only the low-order
-- word of a condition value.  Component accessing functions
-- are provided for SEVERITY, SUCCESS, and MSG_NO for such
-- status values.

subtype WORD_COND_VALUE_TYPE is UNSIGNED_WORD;

function SEVERITY   (STATUS : WORD_COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
function SUCCESS    (STATUS : WORD_COND_VALUE_TYPE)
    return BOOLEAN;
function MSG_NO    (STATUS : WORD_COND_VALUE_TYPE)
    return UNSIGNED_LONGWORD;
```

```
-- Signaling errors.
--
-- The following is an interface routine to LIB$SIGNAL.  This VMS
-- Run-Time Library routine can be used to signal a VMS condition
-- value returned from a VMS system service or Run-Time Library
-- routine.  The resulting exception can be handled in an Ada
-- exception handler using an 'others' exception choice, a handler
-- for the VAX Ada predefined exception NON_ADA_ERROR, or by
-- importing a VMS condition as an Ada exception using the pragma
-- IMPORT_EXCEPTION.
--
--    Usage:
--
--     STATUS : COND_VALUE_TYPE;
--
--     ...
--
--     -- ASSIGN sets STATUS to a condition value.
--     --
--     ASSIGN (STATUS, ...);
--     if not SUCCESS (STATUS) then SIGNAL (STATUS) end if;
--
--    Parameters:
--
--     CV
--          Condition code value
--
--   Notes:
--          See the VMS RTL Library (LIB$) Manual.
--
--
procedure SIGNAL (STATUS : in COND_VALUE_TYPE);
```

```
-- The following is an interface routine to LIB$STOP.  This VMS
-- Run-Time Library routine can be used to signal a VMS condition
-- value returned from a VMS system service or Run-Time Library
-- routine.  The resulting exception can be handled in an Ada
-- exception handler using an 'others' exception choice, a handler
-- for the VAX Ada predefined exception NON_ADA_ERROR, or by
-- importing a VMS condition as an Ada exception using the pragma
-- IMPORT_EXCEPTION.
--
--    Usage:
--
--     STATUS : COND_VALUE_TYPE;
--
--     ...
--
--     -- ASSIGN sets STATUS to a condition value.
--     --
--     ASSIGN (STATUS, ...);
--     if not SUCCESS (STATUS) then STOP (STATUS) end if;
--
--    Parameters:
--
--     CV
--          Condition code value
--
--   Notes:
--          See the VMS RTL Library (LIB$) Manual.
--
--
procedure STOP (STATUS : in COND_VALUE_TYPE);
```

```
-- Matching condition code values.
--
-- The following are interface routines to LIB$MATCH_COND.  This VMS
-- Run-Time Library function can be used to determine if a condition
-- code value matches one or more condition values.
--
--    Usage:
--
--      INDEX := MATCH_COND (CV  : COND_VALUE_TYPE, CV1 : COND_VALUE_TYPE);
--      INDEX := MATCH_COND (CV  : COND_VALUE_TYPE,
--                           CV1 : COND_VALUE_TYPE,
--                           ...   -- for n up to 8
--                           CVn : COND_VALUE_TYPE);
--
--    Parameters:
--
--     CV
--          Value of condition code to match against the list.
--
--     CVi
--          Condition code values, for i from 1 to 8, to be
--          compared to CV.
--
--   Return value:
--
--     INDEX : INTEGER
--          0       If no match found.
--          i - 1   For a match between the first argument and
--                  the ith argument.
--
--   Notes:
--          The algorithm for matching condition codes is described in
--          the VMS RTL Library (LIB$) Manual.
--
function MATCH_COND (
    COND_VALUE   : COND_VALUE_TYPE;
    COND_VALUE_1 : COND_VALUE_TYPE)
    return INTEGER;

function MATCH_COND (
    COND_VALUE   : COND_VALUE_TYPE;
    COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE)
    return INTEGER;

function MATCH_COND (
    COND_VALUE   : COND_VALUE_TYPE;
    COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE;
    COND_VALUE_3 : COND_VALUE_TYPE)
    return INTEGER;
```

```
    function MATCH_COND (
        COND_VALUE   : COND_VALUE_TYPE;
        COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE;
        COND_VALUE_3 : COND_VALUE_TYPE; COND_VALUE_4 : COND_VALUE_TYPE)
        return INTEGER;

    function MATCH_COND (
        COND_VALUE   : COND_VALUE_TYPE;
        COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE;
        COND_VALUE_3 : COND_VALUE_TYPE; COND_VALUE_4 : COND_VALUE_TYPE;
        COND_VALUE_5 : COND_VALUE_TYPE)
        return INTEGER;

    function MATCH_COND (
        COND_VALUE   : COND_VALUE_TYPE;
        COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE;
        COND_VALUE_3 : COND_VALUE_TYPE; COND_VALUE_4 : COND_VALUE_TYPE;
        COND_VALUE_5 : COND_VALUE_TYPE; COND_VALUE_6 : COND_VALUE_TYPE)
        return INTEGER;

    function MATCH_COND (
        COND_VALUE   : COND_VALUE_TYPE;
        COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE;
        COND_VALUE_3 : COND_VALUE_TYPE; COND_VALUE_4 : COND_VALUE_TYPE;
        COND_VALUE_5 : COND_VALUE_TYPE; COND_VALUE_6 : COND_VALUE_TYPE;
        COND_VALUE_7 : COND_VALUE_TYPE)
        return INTEGER;

    function MATCH_COND (
        COND_VALUE   : COND_VALUE_TYPE;
        COND_VALUE_1 : COND_VALUE_TYPE; COND_VALUE_2 : COND_VALUE_TYPE;
        COND_VALUE_3 : COND_VALUE_TYPE; COND_VALUE_4 : COND_VALUE_TYPE;
        COND_VALUE_5 : COND_VALUE_TYPE; COND_VALUE_6 : COND_VALUE_TYPE;
        COND_VALUE_7 : COND_VALUE_TYPE; COND_VALUE_8 : COND_VALUE_TYPE)
        return INTEGER;

private

    -- implementation-defined

end CONDITION_HANDLING;
```

B.4 Package CONTROL_C_INTERCEPTION

```
package CONTROL_C_INTERCEPTION is

    -- This package establishes a CTRL/C handler when it is
    -- elaborated.

    -- If your tasking program fails to stop when you type CTRL/Y
    -- EXIT, or fails to invoke the VMS debugger when you type
    -- CTRL/Y DEBUG, using this package can cure the problem.  Name
    -- the package in a 'with' clause, and elaborate it before
    -- creating tasks in the program.
    --
    -- Example:
    --     WITH CONTROL_C_INTERCEPTION;
    --     pragma ELABORATE(CONTROL_C_INTERCEPTION);
    --     procedure MY_MAIN_PROGRAM is ...
    --
    -- Then, if CTRL/Y does not work as a way to invoke the
    -- debugger or to exit, type CTRL/C. Typing CTRL/C will invoke
    -- the VAX Ada CTRL/C interceptor, which will give you most of
    -- the same options that CTRL/Y would.
    --
    -- NOTE: Other CTRL/C handlers that might be present in the
    -- program will also be able to execute (provided they are also
    -- outband AST handlers), either before or after the VAX Ada
    -- CTRL/C interceptor gives its prompt.

    -- CAUTION: Use of this package can override the intentions of
    -- the DCL SET NOCONTROL_Y command.  It allows a user to type
    -- CTRL/C and get the same degree of control over the program.
    -- Hence, any Ada program that uses this package should not be
    -- used in captive command files (command files that prevent a
    -- user from gaining control by typing CTRL/Y).

end CONTROL_C_INTERCEPTION;
```

B.5 Package MATH_LIB

```
generic
    type REAL is digits <>;
package MATH_LIB is
    pragma IDENT ("VAX Ada Version 2.0");

    -- Square root.
    --
    function SQRT   (A : REAL) return REAL;

    -- Natural logarithm - log base e (A).
    --
```

```
function LOG    (A : REAL) return REAL;

-- Common logarithm - log base 10 (A).
--
function LOG10  (A : REAL) return REAL;

-- Base 2 logarithm - log base 2 (A).  *** not in FORTRAN
--
function LOG2   (A : REAL) return REAL;

-- Exponential.
--
function EXP    (A : REAL) return REAL;

-- Sine, cosine, and tangent of an angle given in radians.
--
function SIN    (A : REAL) return REAL;
function COS    (A : REAL) return REAL;
function TAN    (A : REAL) return REAL;

-- Arc sine, arc cosine, and arc tangent - return an angle
-- expressed in radians.
--
function ASIN   (A : REAL) return REAL;
function ACOS   (A : REAL) return REAL;
function ATAN   (A : REAL) return REAL;

-- Arc tangent with two parameters - Arc Tan (A1/A2) - returns
-- an angle expressed in radians.
--
function ATAN2  (A1, A2 : REAL) return REAL;

-- Hyperbolic sine, cosine, and tangent of an angle in radians.
--
function SINH   (A : REAL) return REAL;
function COSH   (A : REAL) return REAL;
function TANH   (A : REAL) return REAL;

-- Trigonometric functions for angles expressed in degrees.
--
function SIND   (A : REAL) return REAL;
function COSD   (A : REAL) return REAL;
function TAND   (A : REAL) return REAL;
function ASIND  (A : REAL) return REAL;
function ACOSD  (A : REAL) return REAL;
function ATAND  (A : REAL) return REAL;
function ATAN2D (A1, A2 : REAL) return REAL;

-- Exponentiation.
--
function "**" (A1, A2 : REAL) return REAL;

-- Exceptions:  The following exceptions are raised by various
-- VMS Run-Time Library mathematics routines.  See the VMS RTL
-- Mathematics (MTH$) Manual for details.
```

```
    ROPRAND : exception; -- Reserved operand fault.
    INVARGMAT : exception; -- Invalid argument.
    FLOOVEMAT : exception; -- Floating-point overflow in Math Library.
    FLOUNDMAT : exception; -- Floating-point underflow in Math Library.
    LOGZERNEG : exception; -- Logarithm of zero or negative value.
    SQUROONEG : exception; -- Square root of a negative number.

private

    -- implementation defined

end MATH_LIB;

pragma INLINE_GENERIC (MATH_LIB);
```

B.6 Package STARLET

The specification of the types declared in this package follows.

```
-- Ada parameter types and subtypes for VMS system service calls.
--
-- Types defined in the predefined package STANDARD that are used
-- as parameter types for VMS system service calls include
-- BOOLEAN, INTEGER, SHORT_INTEGER, and STRING.  Several types in
-- the predefined package SYSTEM are used, including ADDRESS,
-- AST_HANDLER, UNSIGNED_LONGWORD, and other unsigned types.
--
-- Additional parameter types are defined as follows:

-- DESCRIPTOR_TYPE
--
-- Record structure that is the prototype for a VAX descriptor.
--
type DESCRIPTOR_TYPE is
   record
      LENGTH: SYSTEM.UNSIGNED_WORD;
      DTYPE:  SYSTEM.UNSIGNED_BYTE;
      CLASS:  SYSTEM.UNSIGNED_BYTE;
      POINTER: SYSTEM.ADDRESS;
  end record;

type DESCRIPTOR_ARRAY_TYPE is
  array (NATURAL range <>) of DESCRIPTOR_TYPE;
```

```
-- ACCESS_BIT_NAMES_TYPE
--
-- Array of 32 elements in which each element is a CLASS_S string
-- descriptor; each descriptor names one of the 32 bits in an
-- access mask.  The ACCNAM parameter of the FORMAT_ACL system
-- service is of this type.
--
subtype ACCESS_BIT_NAMES_TYPE is
   STARLET.DESCRIPTOR_ARRAY_TYPE (0..31);


-- ACCESS_MODE_TYPE
--
-- Hardware access mode.  This can take four values: PSL_C_KERNEL
-- (0) specifies kernel mode; PSL_C_EXEC (1) specifies executive mode;
-- PSL_C_SUPER (2) specifies supervisor mode; and PSL_C_USER (3)
-- specifies user mode.
--
subtype ACCESS_MODE_TYPE is
   SYSTEM.UNSIGNED_WORD range 0 .. 3;
ACCESS_MODE_ZERO : constant ACCESS_MODE_TYPE := 0;


-- ADDRESS_RANGE_TYPE
--
-- Array of addresses denoting a range of virtual addresses, which
-- identify an area of memory.  The first address specifies the
-- beginning address in the range; the second specifies the ending
-- address in the range.
--
type ADDRESS_RANGE_TYPE is
   array (0 .. 1) of SYSTEM.ADDRESS;


-- ARG_LIST_TYPE
--
-- Procedure argument list consisting of one or more longwords;
-- the first longword contains an unsigned integer count of the
-- number of successive, contiguous longwords.  Each subsequent
-- longword is a parameter to be passed to a procedure by means of
-- a VAX CALL instruction.
--
subtype ARG_LIST_TYPE is SYSTEM.UNSIGNED_LONGWORD_ARRAY;

-- AST_PROCEDURE_TYPE
--
-- Address of the entry mask to a procedure to be called at AST
-- level.  (Procedures that are not to be called at AST level are
-- called PROCEDURE_TYPE.)
--
subtype AST_PROCEDURE_TYPE is SYSTEM.ADDRESS;
```

```
-- CHANNEL_TYPE
--
-- Unsigned word integer that is an index to an input-output channel.
--
subtype CHANNEL_TYPE is SYSTEM.UNSIGNED_WORD;
CHANNEL_ZERO : constant CHANNEL_TYPE := 0;


-- COND_VALUE_TYPE (CONDITION_HANDLING.COND_VALUE_TYPE)
--
-- VMS condition value.  (See the package CONDITION_HANDLING.)
--
COND_VALUE_ZERO : constant
   CONDITION_HANDLING.COND_VALUE_TYPE := 0;
COND_VALUE_1 : constant
   CONDITION_HANDLING.COND_VALUE_TYPE := 1;


-- CONTEXT_TYPE
--
-- Longword value that is used by a called procedure to maintain
-- some context.
--
subtype CONTEXT_TYPE is SYSTEM.UNSIGNED_LONGWORD;


-- DATE_TIME_TYPE
--
-- 64-bit unsigned binary integer denoting a date and time
-- representing the number of elapsed 100-nanosecond units since
-- 00:00 o'clock, November 17, 1858.
--
subtype DATE_TIME_TYPE is SYSTEM.UNSIGNED_QUADWORD;


-- DEVICE_NAME_TYPE
--
-- Character string denoting the name of a device.  It can be a
-- logical name, but if it is, it must translate to a valid device
-- name.
--
subtype DEVICE_NAME_TYPE is STRING;


-- EF_CLUSTER_NAME_TYPE
--
-- Character string denoting the name of an event flag cluster.
-- It can be a logical name, but if it is, it must translate to a
-- valid event flag cluster name.
--
subtype EF_CLUSTER_NAME_TYPE is STRING;
```

```
-- EF_NUMBER_TYPE
--
-- Unsigned longword integer denoting the number of an event flag.
--
subtype EF_NUMBER_TYPE is SYSTEM.UNSIGNED_LONGWORD;
EF_NUMBER_ZERO : constant EF_NUMBER_TYPE := 0;


-- EXIT_HANDLER_BLOCK_TYPE
--
-- Variable-length structure denoting an exit handler control
-- block.  The DESBLK parameter of the DCLEXH system service is of
-- this subtype.
--
subtype EXIT_HANDLER_BLOCK_TYPE is
   SYSTEM.UNSIGNED_LONGWORD_ARRAY;


-- FAB_TYPE
--
-- VMS RMS file access block.  Type definition is in the package
-- STARLET.


-- FILE_PROTECTION_TYPE
--
-- 16-bit file protection mask.  The mask contains four 4-bit
-- fields, each of which specifies the protection to be applied to
-- file access attempts by one of the four categories of user.
--
subtype FILE_PROTECTION_TYPE is SYSTEM.UNSIGNED_WORD;
FILE_PROTECTION_ZERO : constant
   FILE_PROTECTION_TYPE := 0;

type FILE_PROTECTION_FLAGS_TYPE is
   record
      NOREAD  : BOOLEAN;   -- Deny read access.
      NOWRITE : BOOLEAN;   -- Deny write access.
      NOEXE   : BOOLEAN;   -- Deny execution access.
      NODEL   : BOOLEAN;   -- Deny delete access.
   end record;

pragma PACK (FILE_PROTECTION_FLAGS_TYPE);

type FILE_PROTECTION_REC_TYPE is
   record
      SYS : FILE_PROTECTION_FLAGS_TYPE;
      OWN : FILE_PROTECTION_FLAGS_TYPE;
      GRP : FILE_PROTECTION_FLAGS_TYPE;
      WLD : FILE_PROTECTION_FLAGS_TYPE;
   end record;
```

```
   for FILE_PROTECTION_REC_TYPE use
      record
         SYS at 0 range 0 .. 3;
         OWN at 0 range 4 .. 7;
         GRP at 1 range 0 .. 3;
         WLD at 1 range 4 .. 7;
      end record;

   for FILE_PROTECTION_REC_TYPE'SIZE use 16;


-- FUNCTION_CODE_TYPE
--
-- Unsigned word specifying the operations a system service is to
-- perform.  The FUNC argument to the QIO system service is of
-- this type.  See also ITEM_LIST_3_TYPE.
--
subtype FUNCTION_CODE_TYPE is SYSTEM.UNSIGNED_WORD;
FUNCTION_CODE_ZERO : constant FUNCTION_CODE_TYPE := 0;

-- IDENTIFIER_TYPE
--
-- Unsigned longword that identifies an object returned by the
-- system.
--
subtype IDENTIFIER_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- IO_STATUS_BLOCK_TYPE
--
-- 8-byte record containing information returned by a system
-- service that completes asynchronously.  The information
-- returned varies depending on the service.  For example, see
-- the system services QIO, GETDVI, and GETJPI.
--
type IO_STATUS_BLOCK_TYPE is
   record
      STATUS   : CONDITION_HANDLING.WORD_COND_VALUE_TYPE;
      COUNT    : SYSTEM.UNSIGNED_WORD;
      DEV_INFO : SYSTEM.UNSIGNED_LONGWORD;
   end record;
subtype IOSB_TYPE is IO_STATUS_BLOCK_TYPE;  -- For VAX Ada Version 1
                                            -- compatibility.

-- ITEM_LIST_2_TYPE
--
-- Array of one or more item descriptors that is terminated by a
-- longword containing 0.  Each item descriptor is a 2-longword
-- structure that contains three fields.  The VALUELST parameter
-- of the FILESCAN system service is of this subtype.
--
type ITEM_REC_2_TYPE is
   record
```

```
-- A word in which the service writes the length (in
-- characters) of the requested component.  If the service
-- does not locate the component, it writes the value
-- 0 in this field and in the COMPONENT_ADDRESS field and
-- returns the SS_NORMAL condition code.
--
      COMPONENT_LENGTH : SYSTEM.UNSIGNED_WORD;

-- A user-supplied, word-length symbolic code that specifies
-- the component desired.
--
      ITEM_CODE    : STARLET.FUNCTION_CODE_TYPE;

-- A longword in which the service writes the starting
-- address of the component.  This address points to a
-- location in the input string itself.
--
      COMPONENT_ADDRESS : SYSTEM.ADDRESS;

end record;

type ITEM_LIST_2_TYPE is
   array(NATURAL range <>) of ITEM_REC_2_TYPE;


-- ITEM_LIST_3_TYPE
--
-- Array that consists of one or more item descriptors and that is
-- terminated by a longword containing 0.  Each item descriptor is
-- a 3-longword structure that contains four fields.  The ITMLST
-- parameter of the GETDVI system service is of this type.  Also
-- known as ITEM_LIST_TYPE.
--
type ITEM_REC_TYPE is
   record

      -- Length of the buffer (in bytes) containing or
      -- receiving the information specified by
      -- ITEM_CODE.
      --
      BUF_LEN     : SYSTEM.UNSIGNED_WORD;

      -- Code indicating the operation to be performed.
      --
      ITEM_CODE    : STARLET.FUNCTION_CODE_TYPE;

      -- Address of the buffer containing or receiving
      -- the information specified by ITEM_CODE.
      --
      BUF_ADDRESS : SYSTEM.ADDRESS;

      -- Address of a word to receive the length of the
      -- information returned.
      --
      RET_ADDRESS : SYSTEM.ADDRESS;

end record;
```

```
type ITEM_LIST_3_TYPE is
   array(NATURAL range <>) of ITEM_REC_TYPE;
subtype ITEM_LIST_TYPE is ITEM_LIST_3_TYPE;


-- ITEM_LIST_PAIR_TYPE
--
-- Structure that consists of one or more longword pairs, or
-- doublets, and is terminated by a longword containing 0.
-- Typically, the first longword contains an integer value such as
-- a code.  The second longword can contain a real or integer
-- value.
--
type ITEM_LIST_PAIR_REC_TYPE is
   record
      L0: SYSTEM.UNSIGNED_LONGWORD;
      L1: SYSTEM.UNSIGNED_LONGWORD;
   end record;

type ITEM_LIST_PAIR_TYPE is
   array (NATURAL range <>) of ITEM_LIST_PAIR_REC_TYPE;


-- ITEM_QUOTA_LIST_TYPE
--
-- Array of one or more quota item descriptors that is terminated
-- by an item code of 0.  Each quota item descriptor is an
-- unsigned byte code followed by a longword value for that quota
-- item.  The QUOTA parameter of the CREPRC system service is of
-- this type.
--

type ITEM_QUOTA_REC_TYPE is
   record

      -- Code indicating the quota to be assigned.  The end of the
      -- list is designated by an item code of PQL_LISTEND.
      --
      ITEM_CODE : SYSTEM.UNSIGNED_BYTE;

      -- Value of the quota to be assigned.
      --
      ITEM_VALUE : SYSTEM.UNSIGNED_LONGWORD;

end record;
type ITEM_QUOTA_LIST_TYPE is
   array (NATURAL range <>) of ITEM_QUOTA_REC_TYPE;
```

```
-- LOCK_ID_TYPE
--
-- Longword value denoting a lock identifier.  This lock
-- identifier is assigned by the lock manager facility to a lock
-- when the lock is granted.
--
type LOCK_ID_TYPE is new SYSTEM.UNSIGNED_LONGWORD;
LOCK_ID_ZERO : constant LOCK_ID_TYPE := 0;


-- LOCK_VALUE_BLOCK_TYPE
--
-- 16-byte block that the lock manager facility includes in a lock
-- status block if the user requests it; the contents of the lock
-- value block are user-defined and are not interpreted by the
-- lock manager facility.
--
type LOCK_VALUE_BLOCK_TYPE is
   new SYSTEM.UNSIGNED_BYTE_ARRAY (1 .. 16);


-- LOCK_STATUS_BLOCK_TYPE
--
-- Structure into which the lock manager facility writes status
-- information about a lock.  A lock status block always contains
-- at least two longwords: the first word of the first longword
-- contains a status code; the second word of the first longword
-- is reserved by Digital; and the second longword contains the
-- lock identifier.  In addition to these fields, a lock status
-- block may optionally include a 16-byte lock value block.
--
type LOCK_STATUS_BLOCK_TYPE is
   record
      STATUS   : CONDITION_HANDLING.WORD_COND_VALUE_TYPE;
      RESERVED : SYSTEM.UNSIGNED_WORD;
      ID       : STARLET.LOCK_ID_TYPE;
      VALU     : STARLET.LOCK_VALUE_BLOCK_TYPE;
   end record;


-- LOGICAL_NAME_TYPE
--
-- Character string of from 1 to 255 characters that identifies a
-- logical name or equivalence name to be manipulated by VMS
-- logical name system services.
--
subtype LOGICAL_NAME_TYPE is STRING;
```

```
   -- MASK_PRIVILEGES_TYPE
   --
   -- 64-bit record wherein each individual bit denotes a process
   -- privilege.  The PRVADR parameter of the CREPRC system service
   -- is of this subtype.
   --
   subtype MASK_PRIVILEGES_TYPE is STARLET.PRV_TYPE;


   -- PAGE_PROTECTION_TYPE
   --
   -- Longword value specifying the page protection to be applied by
   -- the VAX hardware.  The PROT parameter of the SETPRT system
   -- service is of this subtype.  Symbolic values for page
   -- protection are PRT_C_xxx.
   --
   subtype PAGE_PROTECTION_TYPE is SYSTEM.UNSIGNED_LONGWORD;


   -- PROCEDURE_TYPE
   --
   -- Address of the entry mask to a procedure that is not to be
   -- called at AST level.  (Arguments specifying procedures to be
   -- called at AST level have the type AST_PROCEDURE_TYPE.)
   --
   subtype PROCEDURE_TYPE is SYSTEM.ADDRESS;


   -- PROCESS_ID_TYPE
   --
   -- Longword value denoting a process identifier (PID).  This
   -- process identifier is assigned by the VMS operating system to
   -- a process when the process is created.  The PIDADR parameter
   -- of the DELPRC system service is of this subtype.
   --
   subtype PROCESS_ID_TYPE is SYSTEM.UNSIGNED_LONGWORD;


   -- PROCESS_NAME_TYPE
   --
   -- Character string that specifies the name of a process.
   --
   subtype PROCESS_NAME_TYPE is STRING;


   -- RIGHTS_ID_TYPE
   --
   -- Longword value denoting a rights identifier, which identifies
   -- an interest group in the context of the VMS security
   -- environment.  This rights identifier may consist of all or part
   -- of a user's User Identification Code (UIC).
   --
   subtype RIGHTS_ID_TYPE is SYSTEM.UNSIGNED_LONGWORD;
   RIGHTS_ID_ZERO : constant RIGHTS_ID_TYPE := 0;
```

```
-- RIGHTS_HOLDER_TYPE
--
-- 64-bit record specifying a user's access rights to a system
-- object.  The RACCESS component is a longword bit mask wherein
-- each bit specifies an access right.  The HOLDER parameter of
-- the ADD_HOLDER system service is of this type.
--
type RIGHTS_HOLDER_TYPE is
   record
      RIGHTS_ID : STARLET.RIGHTS_ID_TYPE;
      RACCESS   : STARLET.KGB_ATTRIBUTES_TYPE;
   end record;


-- RAB_TYPE
--
-- VMS RMS record access block.  Type definition is in the package
-- STARLET.


-- RU_HANDLE_TYPE
--
-- Longword value that is used by the recovery unit services to
-- identify a particular recovery unit.
--
type RU_HANDLE_TYPE is new SYSTEM.UNSIGNED_LONGWORD;

-- SECTION_ID_TYPE
--
-- Quadword value denoting a global section identifier.  This
-- identifier specifies the version of a global section and the
-- criteria to be used in matching that global section.  The IDENT
-- parameter of the MGBLSC system service is of this subtype.
--
type SECTION_ID_TYPE is new SYSTEM.UNSIGNED_QUADWORD;


-- SECTION_NAME_TYPE
--
-- Character string denoting a global section name.  This
-- character string can be a logical name, but it must translate
-- to a valid global section name.  The GSDNAM parameter of the
-- MGBLSC system service is of this subtype.
--
subtype SECTION_NAME_TYPE is STRING;
```

```
   -- SYSTEM_ACCESS_ID_TYPE
   --
   -- Quadword value that denotes a system identification value that
   -- is to be associated with a rights database.  The SYSID
   -- parameter of the CREATE_RDB system service is of this subtype.
   --
   subtype SYSTEM_ACCESS_ID_TYPE is
      SYSTEM.UNSIGNED_QUADWORD;


   -- TIME_NAME_TYPE
   --
   -- Character string specifying a time value in VMS format.  The
   -- TIMBUF parameter of the ASCTIM system service is of this
   -- subtype.
   --
   subtype TIME_NAME_TYPE is STRING;

   -- TRANSACTION_ID_TYPE
   --
   -- 16-byte value that uniquely identifies a transaction.
   --
   type TRANSACTION_ID_TYPE is
      new SYSTEM.UNSIGNED_BYTE_ARRAY (0..15);

   -- UIC_TYPE
   --
   -- Longword value denoting a User Identification Code (UIC).
   --
   subtype UIC_LONGWORD_TYPE is SYSTEM.UNSIGNED_LONGWORD;
   UIC_LONGWORD_ZERO : constant UIC_LONGWORD_TYPE := 0;


   -- USER_ARG_TYPE
   --
   -- Longword value denoting a user-defined argument.  This longword
   -- is passed to a procedure as a parameter, but the contents of
   -- the longword are defined and interpreted by the user.  The
   -- ASTPRM parameter of the QIO system service is of this subtype.
   --
   subtype USER_ARG_TYPE is SYSTEM.UNSIGNED_LONGWORD;
   USER_ARG_ZERO : constant USER_ARG_TYPE := 0;

   -- The following additional type names are used in the VMS
   -- documentation and are aliases of existing Ada types.  They are
   -- defined here for completeness and ease of use.
   --
   subtype BYTE_SIGNED_TYPE is SHORT_SHORT_INTEGER;
   subtype BYTE_UNSIGNED_TYPE is SYSTEM.UNSIGNED_BYTE;

   subtype CHAR_STRING_TYPE is STRING;
```

```
   subtype LONGWORD_SIGNED_TYPE is INTEGER;
   subtype LONGWORD_UNSIGNED_TYPE is SYSTEM.UNSIGNED_LONGWORD;

   subtype MASK_BYTE_TYPE is SYSTEM.UNSIGNED_BYTE;
   subtype MASK_LONGWORD_TYPE is SYSTEM.UNSIGNED_LONGWORD;
   subtype MASK_QUADWORD_TYPE is SYSTEM.UNSIGNED_QUADWORD;
   subtype MASK_WORD_TYPE is SYSTEM.UNSIGNED_WORD;

   type VECTOR_BYTE_SIGNED_TYPE is
     array (NATURAL range <>) of SHORT_SHORT_INTEGER;
   subtype VECTOR_BYTE_UNSIGNED_TYPE is
     SYSTEM.UNSIGNED_BYTE_ARRAY;

   type VECTOR_LONGWORD_SIGNED_TYPE is
     array (NATURAL range <>) of INTEGER;
   subtype VECTOR_LONGWORD_UNSIGNED_TYPE is
     SYSTEM.UNSIGNED_LONGWORD_ARRAY;

   type VECTOR_WORD_SIGNED_TYPE is
     array (NATURAL range <>) of SHORT_INTEGER;
   subtype VECTOR_WORD_UNSIGNED_TYPE is
     SYSTEM.UNSIGNED_WORD_ARRAY;
```

B.7 Package SYSTEM_RUNTIME_TUNING

```
-- This package defines interfaces to allow user programs to
-- change various parameters that affect Ada program execution
-- and that are normally chosen by the VAX Ada run-time library.
--

package SYSTEM_RUNTIME_TUNING is

    subtype AST_PACKET_REQUEST_TYPE is NATURAL range 0 .. 1_048_576;

    procedure EXPAND_AST_PACKET_POOL (
        REQUESTED_PACKETS       : in AST_PACKET_REQUEST_TYPE;
        ACTUAL_NUMBER           : out NATURAL;
        TOTAL_NUMBER            : out NATURAL);
```

```
-- FUNCTIONAL DESCRIPTION
--
-- This routine adds more AST packets to the pool of packets used
-- by the AST_ENTRY attribute.  It supports the creation of up
-- to 1048576 packets.  The success of the call depends on there
-- being enough virtual memory to satisfy the request.  (A single
-- AST packet currently consumes 32 bytes of dynamic memory.)
--
-- When you use the AST_ENTRY attribute to handle an AST, an AST
-- packet is used by the Ada run-time library to hold the AST
-- parameter.  An AST packet is in use from the time when the AST
-- is delivered by the VMS operating system until the receiving
-- task completes the accept statement receiving the AST
-- parameter.  If the peak number of ASTs delivered by the VMS
-- operating system, but not yet accepted, exceeds the size of the
-- AST packet pool, an unrecoverable error occurs, stating that
-- the AST packet pool has been exhausted.  The
-- EXPAND_AST_PACKET_POOL routine can help eliminate that error by
-- increasing the size of the AST packet pool.
--
-- Before increasing the AST packet pool, try to minimize the peak
-- number of AST packets required by your program.  To do this, try
-- to ensure that the accepting task has a very high priority, and
-- is not delayed by an interaction with any other task before or
-- during the accept statement for the AST.  Only after you have
-- concluded that the AST arrival rate is so high that it
-- momentarily exceeds your program's rate of servicing the ASTs
-- should you consider using this routine to increase the size of
-- the pool.
--
-- NOTE: Using this routine will not help if your program's average
-- AST arrival rate is greater than its average AST service rate,
-- because eventually your program will still run out of AST
-- packets.  In this case, you need to revise your program to reduce
-- the AST arrival rate -- how you do that depends on your
-- application.
--
-- INPUT PARAMETERS:
--
-- REQUESTED_PACKETS is the minimum number of additional packets
-- desired.  More may be allocated because of rounding to the next
-- storage boundary.  To determine the current size of the pool, you
-- can specify 0 for REQUESTED_PACKETS.
--
-- OUTPUT PARAMETERS:
--
-- ACTUAL_NUMBER indicates the number of packets that were added to
-- the pool.
--
-- TOTAL_NUMBER indicates the total number of AST packets in the
-- pool.  (Note that this number includes AST packets that might be
-- currently in use for the delivery of an AST.)
--
```

```
--
-- EXCEPTIONS:
-- STORAGE_ERROR is raised if the request could not be satisfied
-- because of insufficient memory. When STORAGE_ERROR is raised, an
-- attempt is made to release any AST packets allocated in partial
-- fulfillment of the request. PROGRAM_ERROR may be raised for
-- certain other errors. If PROGRAM_ERROR is raised, a chained
-- condition indicates a detailed reason for the failure.  Other
-- exceptions may be raised as well.
--

pragma INTERFACE(RTL, EXPAND_AST_PACKET_POOL);
pragma IMPORT_PROCEDURE(EXPAND_AST_PACKET_POOL,
    "ADA$EXPAND_AST_PACKET_POOL");


procedure REQUEST_TIME_SLICE (REQUESTED_VALUE :     DURATION);

-- FUNCTIONAL_DESCRIPTION:
--
-- This routine conditionally modifies the time-slice setting of the
-- program. This entry point can only make time slicing run faster
-- than it is already running, or enable it if it is not enabled.
-- The request will be overridden by the value specified by a pragma
--  TIME_SLICE in an Ada main program or by a debugger SET TASK
-- /TIME_SLICE command.
--
-- This routine is primarily intended to be called from within an
-- Ada shareable image or an object file exported by an ACS EXPORT
-- command, where it cannot be decided in advance whether there will
-- be an Ada main program.  However, this routine can also be used
-- to override the wishes of an Ada main program that does not
-- specify a pragma TIME_SLICE (and as often as desired).
--
-- This call has no effect if any of the following are true:
--
--     1. REQUESTED_VALUE is 0.0 or negative (time slicing
--        cannot be disabled by this routine).
--
--     2. REQUESTED_VALUE is greater than a previously specified
--        time-slice value that successfully set the time slice.
--
--     3. Time slicing has either been activated or turned off
--        by a pragma TIME_SLICE.
--
--     4. A debugger SET TASK/TIME_SLICE=t command has been
--        issued.
--
-- If none of these conditions is true, then REQUESTED_VALUE will
-- set the time slice.
--
-- In the following cases, the time slice set by this call will be
-- overridden:
--
```

```
    --      1. An image containing an Ada main program that has a pragma
    --         TIME_SLICE is activated.
    --
    --      2. A debugger TASK/TIME_SLICE=t command is issued.
    --
    --      3. REQUEST_TIME_SLICE is called again with a REQUESTED_VALUE
    --         greater than 0 but less than the value set by this call.
    --
    -- INPUT PARAMETERS:
    --
    -- REQUESTED_VALUE is the requested new time-slice value.
    --
    -- EXCEPTIONS:
    -- PROGRAM_ERROR may be raised for certain errors.  If PROGRAM_ERROR
    -- is raised, a chained condition indicates a detailed reason for
    -- the failure.  Other exceptions may be raised as well.

    pragma INTERFACE(RTL, REQUEST_TIME_SLICE);
    pragma IMPORT_PROCEDURE(REQUEST_TIME_SLICE,
        "ADA$SET_TIME_SLICE");

end SYSTEM_RUNTIME_TUNING;
```

B.8 Package TASKING_SERVICES

```
with STARLET; use STARLET;
with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with RMS_ASYNCH_OPERATIONS; use RMS_ASYNCH_OPERATIONS;
package TASKING_SERVICES is
    pragma IDENT ("VAX Ada Version 2.0");
```

```
-- DESCRIPTION:
--
-- Certain VMS system services allow the calling routine to
-- synchronize with the completion of the service. In other
-- words, after issuing (queuing) the system service call,
-- the calling routine may continue executing until the
-- service completes. Then, the routine can synchronize with
-- the service using an event flag and/or an AST handler.
--
-- If one of these services is called by a VAX Ada task in
-- a program where time slicing has not been enabled, and
-- event-flag synchronization has been chosen (for example,
-- SYS$QIOW has been called), the process in which the
-- calling task is executing will be suspended until the
-- event flag is set. This effect may not be desired
-- because suspension of the process causes all other
-- executing tasks to be suspended, even if they are ready
-- for execution.
--
-- This package provides a convenient interface to VMS
-- system services that have both synchronous and
-- asynchronous forms (for example, SYS$QIOW and SYS$QIO).
-- The package is designed for those situations in which
-- you would like a particular task to wait for the
-- completion of a system service, but do not want to
-- prevent other tasks in your program from executing.
-- Although the pragma AST_ENTRY and attribute T'AST_ENTRY
-- provide a more general way of achieving the same effect,
-- they require more detailed programming.
--
-- This package provides an interface to the following VMS
-- system services:
--
--      System service          TASKING_SERVICES name
--
--          $BRKTHRU                TASK_BRKTHRUW
--          $ENQ                    TASK_ENQW
--          $GETDVI                 TASK_GETDVIW
--          $GETJPI                 TASK_GETJPIW
--          $GETLKI                 TASK_GETLKIW
--          $GETQUI                 TASK_GETQUIW
--          $GETSYI                 TASK_GETSYIW
--          $QIO                    TASK_QIOW
--          $SNDJBC                 TASK_SNDJBCW
--          $UPDSEC                 TASK_UPDSECW
--          all RMS operations      TASK_RMS_*
--                                  (for example,
--                                   TASK_RMS_GET)
--
--
-- The signatures of the subprogram declarations
-- generally correspond to those in the package STARLET,
-- and use the same parameter types declared in the
```

```
-- package STARLET.  Refer to Chapter 6 of the VAX Ada
-- Run-Time Reference Manual for information on calling
-- system services; the same considerations apply for
-- both the packages STARLET and TASKING_SERVICES.
-- Differences from the declarations in the package
-- STARLET are:
--
--   1. The AST handler and AST parameter parameters are
--      omitted because they are used to implement this
--      package.
--
--   2. In the package STARLET, a default value of
--      type-name'NULL_PARAMETER is used for optional
--      input arguments that are passed by reference or
--      descriptor.  In this package, multiple over-
--      loadings are used to achieve the same effect.

--  TASK_BRKTHRUW
--
--    Write to terminal breakthrough and suspend the
--    task until the system service is completed.
--
--    Parameters:
--
--      efn    = Number of the event flag to be set on completion.
--
--      msgbuf = Address of a message buffer descriptor.
--
--      sendto = Address of a descriptor specifying the receiver of
--      the message.
--
--      sndtyp = Terminal type to which the message is sent.
--
--      iosb   = Address of a quadword input-output status block.
--
--      carcon = Carriage control.
--
--      flags  = Flags to modify the broadcast.
--
--      reqid  = Broadcast class requestor identification.
--
--      timout = Address of the timeout value.
--
--
procedure TASK_BRKTHRUW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF  : in  STRING;
    SENDTO  : in  STRING;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
    IOSB    : out IO_STATUS_BLOCK_TYPE;
    CARCON  : in  UNSIGNED_LONGWORD := 32;
    FLAGS   : in  BRK_TYPE;
```

```
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    IGNORED  : in  ADDRESS           := ADDRESS_ZERO;
    SENDTO   : in  STRING;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF   : in  STRING;
    IGNORED  : in  ADDRESS           := ADDRESS_ZERO;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF   : in  STRING;
    SENDTO   : in  STRING;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : in  ADDRESS           := ADDRESS_ZERO;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    IGNORED  : in  ADDRESS           := ADDRESS_ZERO;
    SENDTO   : in  STRING;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : in  ADDRESS           := ADDRESS_ZERO;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
```

```
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF   : in  STRING;
    IGNORED  : in  ADDRESS           := ADDRESS_ZERO;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : in  ADDRESS           := ADDRESS_ZERO;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : in  ADDRESS           := ADDRESS_ZERO;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  BRK_TYPE;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);

--  TASK_ENQW
--
--    Enqueue lock request and suspend the task until the
--    lock is either granted or converted.
--
--    Parameters:
--
--      efn      = Number of the event flag to be set on completion.
--
--      lkmode   = Type of lock mode requested.  Modes are:
--
--                   LCK_K_NLMODE null lock
--                   LCK_K_CRMODE concurrent read
--                   LCK_K_CWMODE concurrent write
--                   LCK_K_PRMODE protected read
--                   LCK_K_PWMODE protected write
--                   LCK_K_EXMODE exclusive lock
--
--      lksb    = Address of the lock status block.
--
--      flags    = Flags defining the characteristics of the
--                   lock.  These are:
--
--                   LCK_M_NOQUEUE
--                   LCK_M_SYNCSTS
--                   LCK_M_SYSTEM
--                   LCK_M_VALBLK
--                   LCK_M_CONVERT
--
```

```
--       resnam    = Address of the string descriptor of the resource
--                   name.
--
--       parid     = Lock identification of the parent lock.
--
--       blkast    = Address of entry mask of the blocking AST
--                   routine; this AST handler must ignore its
--                   AST parameter.
--
--       acmode    = Access mode to be associated with the lock.
--
--       reserved = Reserved for future use.
--
procedure TASK_ENQW (
    STATUS   : out    COND_VALUE_TYPE;
    EFN      : in     EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    LKMODE   : in     UNSIGNED_LONGWORD;
    LKSB     : in out LOCK_STATUS_BLOCK_TYPE;
    FLAGS    : in     LCK_TYPE;
    RESNAM   : in     STRING;
    PARID    : in     LOCK_ID_TYPE            := LOCK_ID_ZERO;
    BLKAST   : in     AST_HANDLER             := NO_AST_HANDLER;
    ACMODE   : in     ACCESS_MODE_TYPE        := ACCESS_MODE_ZERO;
    RESERVED: in      ADDRESS                 := ADDRESS_ZERO);
procedure TASK_ENQW (
    STATUS   : out    COND_VALUE_TYPE;
    EFN      : in     EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    LKMODE   : in     UNSIGNED_LONGWORD;
    LKSB     : in out LOCK_STATUS_BLOCK_TYPE;
    FLAGS    : in     LCK_TYPE;
    PARID    : in     LOCK_ID_TYPE            := LOCK_ID_ZERO;
    BLKAST   : in     AST_HANDLER             := NO_AST_HANDLER;
    ACMODE   : in     ACCESS_MODE_TYPE        := ACCESS_MODE_ZERO;
    RESERVED: in      ADDRESS                 := ADDRESS_ZERO);
```

```
--  TASK_GETDVIW
--
--    Get device/volume information and suspend the task
--    until the operation is complete.
--
--    Parameters:
--
--      efn     = Number of the event flag to be set on completion.
--
--      chan    = Number of a channel assigned to the device or
--                0 if the device is specified by the devnam
--                parameter.
--
--      devnam = Address of the device name or logical name
--                descriptor.
--
--      itmlst = Address of a list of item descriptors.
--
--      iosb    = Address of a quadword input-output status block.
--
--      nullarg= Reserved argument.
--
procedure TASK_GETDVIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CHAN    : in  CHANNEL_TYPE            := CHANNEL_ZERO;
    DEVNAM  : in  DEVICE_NAME_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETDVIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CHAN    : in  CHANNEL_TYPE            := CHANNEL_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETDVIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CHAN    : in  CHANNEL_TYPE            := CHANNEL_ZERO;
    DEVNAM  : in  DEVICE_NAME_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETDVIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CHAN    : in  CHANNEL_TYPE            := CHANNEL_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO);
```

```
--  TASK_GETJPIW
--
--    Get job/process information and suspend the task
--    until the operation is complete.
--
--    Parameters:
--
--      efn    = Number of the event flag to be set on completion.
--
--      pidadr = Address of the process identification.
--
--      prcnam = Address of the process name string descriptor.
--
--      itmlst = Address of a list of item descriptors.
--
--      iosb   = Address of a quadword input-output status block.
--
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in out PROCESS_ID_TYPE;
    PRCNAM  : in  PROCESS_NAME_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in out PROCESS_ID_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in out PROCESS_ID_TYPE;
    PRCNAM  : in  PROCESS_NAME_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in out PROCESS_ID_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    PRCNAM  : in  PROCESS_NAME_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
```

```
    PIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    PRCNAM  : in  PROCESS_NAME_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETJPIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    PIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE);

--  TASK_GETLKIW
--
--    Get lock information and suspend the task until the
--    operation is complete.
--
--    Parameters:
--
--      efn      = Number of the event flag to be set on completion.
--
--      lkidadr  = Address of the lock identification.
--
--      itmlst   = Address of a list of item descriptors.
--
--      iosb     = Address of a quadword input-output status block.
--
--      reserved = Reserved parameter.
--
procedure TASK_GETLKIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    LKIDADR : in out LOCK_ID_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE;
    RESERVED: in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETLKIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    LKIDADR : in out LOCK_ID_TYPE;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO;
    RESERVED: in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETLKIW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    LKIDADR : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
```

```
    IOSB     : in  ADDRESS                  := ADDRESS_ZERO;
    RESERVED: in  ADDRESS                   := ADDRESS_ZERO);
procedure TASK_GETLKIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE           := EF_NUMBER_ZERO;
    LKIDADR  : in  ADDRESS                  := ADDRESS_ZERO;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    RESERVED: in  ADDRESS                   := ADDRESS_ZERO);

--  TASK_GETQUIW
--
--    Get queue information and suspend the task until the
--    operation is complete.
--
--    Parameters:
--
--      efn       = Number of the event flag to be set on completion.
--
--      func      = Code specifying the function to be performed.
--
--      nullarg   = Reserved for future use.
--
--      itmlst    = Address of a list of item descriptors.
--
--      iosb      = Address of a quadword input-output status block.
--
procedure TASK_GETQUIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE           := EF_NUMBER_ZERO;
    FUNC     : in  FUNCTION_CODE_TYPE;
    NULLARG  : in  ADDRESS                  := ADDRESS_ZERO;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETQUIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE           := EF_NUMBER_ZERO;
    FUNC     : in  FUNCTION_CODE_TYPE;
    NULLARG  : in  ADDRESS                  := ADDRESS_ZERO;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : in  ADDRESS                  := ADDRESS_ZERO);
procedure TASK_GETQUIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE           := EF_NUMBER_ZERO;
    FUNC     : in  FUNCTION_CODE_TYPE;
    NULLARG  : in  ADDRESS                  := ADDRESS_ZERO;
    ITMLST   : in  ADDRESS                  := ADDRESS_ZERO;
    IOSB     : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETQUIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE           := EF_NUMBER_ZERO;
    FUNC     : in  FUNCTION_CODE_TYPE;
    NULLARG  : in  ADDRESS                  := ADDRESS_ZERO;
```

```
    ITMLST   : in  ADDRESS                  := ADDRESS_ZERO;
    IOSB     : in  ADDRESS                  := ADDRESS_ZERO);

--  TASK_GETSYIW
--
--    Get system-wide information and suspend the task until
--    the operation is complete.
--
--    Parameters:
--
--      efn       = Number of the event flag to be set on completion.
--
--      csidadr   = Address of the cluster system identification.
--
--      nodename = Address of the node name string descriptor.
--
--      itmlst    = Address of a list of item descriptors.
--
--      iosb      = Address of a quadword input-output status block.
--
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in out PROCESS_ID_TYPE;
    NODENAME: in  PROCESS_NAME_TYPE;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in out PROCESS_ID_TYPE;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in out PROCESS_ID_TYPE;
    NODENAME: in  PROCESS_NAME_TYPE;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in out PROCESS_ID_TYPE;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    NODENAME: in  PROCESS_NAME_TYPE;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : in  ADDRESS                 := ADDRESS_ZERO);
```

```
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : in  ADDRESS                 := ADDRESS_ZERO);
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    NODENAME: in  PROCESS_NAME_TYPE;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE);
procedure TASK_GETSYIW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CSIDADR  : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST   : in  ITEM_LIST_3_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE);

--  TASK_QIOW
--
--    Queue input-output request and suspend the task until
--    input-output is completed.
--
--    Parameters:
--
--      efn     = Number of the event flag to be set on completion.
--
--      chan    = Number of the channel on which input-output is
--                directed.
--
--      func    = Function code specifying the action to be
--                performed.
--
--      iosb    = Address of quadword input-output status block to
--                receive final completion status.
--
--      p1...   = Optional device- and function-specific
--                parameters.
--
--
procedure TASK_QIOW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CHAN     : in  CHANNEL_TYPE;
    FUNC     : in  FUNCTION_CODE_TYPE;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    P1       : in  UNSIGNED_LONGWORD       := 0;
    P2       : in  UNSIGNED_LONGWORD       := 0;
    P3       : in  UNSIGNED_LONGWORD       := 0;
    P4       : in  UNSIGNED_LONGWORD       := 0;
    P5       : in  UNSIGNED_LONGWORD       := 0;
    P6       : in  UNSIGNED_LONGWORD       := 0);
```

```
procedure TASK_QIOW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    CHAN    : in  CHANNEL_TYPE;
    FUNC    : in  FUNCTION_CODE_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO;
    P1      : in  UNSIGNED_LONGWORD       := 0;
    P2      : in  UNSIGNED_LONGWORD       := 0;
    P3      : in  UNSIGNED_LONGWORD       := 0;
    P4      : in  UNSIGNED_LONGWORD       := 0;
    P5      : in  UNSIGNED_LONGWORD       := 0;
    P6      : in  UNSIGNED_LONGWORD       := 0);

--  TASK_SNDJBCW
--
--    Send message to the job controller and suspend the
--    task until the operation completes.
--
--    Parameters:
--
--      efn     = Number of the event flag to be set on completion.
--
--      func    = Code specifying the function to be performed.
--
--      nullarg = Reserved argument for similarity with
--                $GETxxx services.
--
--      itmlst  = Address of a list of item descriptors for
--                the operation.
--
--      iosb    = Address of a quadword input-output status block to
--                receive the final status.
--
procedure TASK_SNDJBCW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    FUNC    : in  FUNCTION_CODE_TYPE;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : out IO_STATUS_BLOCK_TYPE);
procedure TASK_SNDJBCW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    FUNC    : in  FUNCTION_CODE_TYPE;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO;
    IOSB    : out IO_STATUS_BLOCK_TYPE);
procedure TASK_SNDJBCW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    FUNC    : in  FUNCTION_CODE_TYPE;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO;
    ITMLST  : in  ITEM_LIST_3_TYPE;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);
```

```
procedure TASK_SNDJBCW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    FUNC    : in  FUNCTION_CODE_TYPE;
    NULLARG : in  ADDRESS                 := ADDRESS_ZERO;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);

--  TASK_UPDSECW
--
--    Update section file on disk and suspend the task
--    until the operation is complete.
--
--    Parameters:
--
--      inadr  = Address of a 2-longword array containing
--               the starting and ending addresses of the
--               pages  to  be potentially written.
--
--      retadr = Address of a 2-longword array to receive
--               the addresses of the first and last page
--               queued in the first input-output request.
--
--      acmode = Access  mode  on  behalf  of  which
--               the  service  is performed.
--
--      updflg = Update indicator for read/write global
--               sections.
--                 0 -> write all read/write pages
--                      in the section
--                 1 -> write all pages modified by
--                      the caller
--
--      efn    = Number of the event flag to be set when the section
--               file  is updated.
--
--      iosb   = Address of the quadword input-output status block.
--
procedure TASK_UPDSECW (
    STATUS  : out COND_VALUE_TYPE;
    INADR   : in  ADDRESS_RANGE_TYPE;
    RETADR  : out ADDRESS_RANGE_TYPE;
    ACMODE  : in  ACCESS_MODE_TYPE        := ACCESS_MODE_ZERO;
    UPDFLG  : in  BOOLEAN                 := FALSE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    IOSB    : out IO_STATUS_BLOCK_TYPE);
procedure TASK_UPDSECW (
    STATUS  : out COND_VALUE_TYPE;
    INADR   : in  ADDRESS_RANGE_TYPE;
    RETADR  : out ADDRESS_RANGE_TYPE;
    ACMODE  : in  ACCESS_MODE_TYPE        := ACCESS_MODE_ZERO;
    UPDFLG  : in  BOOLEAN                 := FALSE;
    EFN     : in  EF_NUMBER_TYPE          := EF_NUMBER_ZERO;
    IOSB    : in  ADDRESS                 := ADDRESS_ZERO);
```

```
    procedure TASK_UPDSECW (
        STATUS  : out COND_VALUE_TYPE;
        INADR   : in  ADDRESS_RANGE_TYPE;
        RETADR  : in  ADDRESS               := ADDRESS_ZERO;
        ACMODE  : in  ACCESS_MODE_TYPE      := ACCESS_MODE_ZERO;
        UPDFLG  : in  BOOLEAN               := FALSE;
        EFN     : in  EF_NUMBER_TYPE        := EF_NUMBER_ZERO;
        IOSB    : in  ADDRESS               := ADDRESS_ZERO);
    procedure TASK_UPDSECW (
        STATUS  : out COND_VALUE_TYPE;
        INADR   : in  ADDRESS_RANGE_TYPE;
        RETADR  : in  ADDRESS               := ADDRESS_ZERO;
        ACMODE  : in  ACCESS_MODE_TYPE      := ACCESS_MODE_ZERO;
        UPDFLG  : in  BOOLEAN               := FALSE;
        EFN     : in  EF_NUMBER_TYPE        := EF_NUMBER_ZERO;
        IOSB    : out IO_STATUS_BLOCK_TYPE);

    --  TASK_RMS_CLOSE
    --
    --    Close the file.
    --
    --    Parameter:
    --
    --      fab = Address of the file access block.
    --
    procedure TASK_RMS_CLOSE is
        new RMS_ASYNCH_FAB_OPERATION (STARLET.CLOSE);

    --  TASK_RMS_CONNECT
    --
    --    Connect the file.
    --
    --    Parameters:
    --
    --      rab = Address of the record access block.
    --
    procedure TASK_RMS_CONNECT is
        new RMS_ASYNCH_RAB_OPERATION (STARLET.CONNECT);

    --  TASK_RMS_CREATE
    --
    --    Create the file.
    --
    --    Parameters:
    --
    --      fab = Address of the file access block.
    --
    procedure TASK_RMS_CREATE is
        new RMS_ASYNCH_FAB_OPERATION (STARLET.CREATE);
```

```
--  TASK_RMS_DELETE
--
--    Delete the record.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_DELETE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.DELETE);

--  TASK_RMS_DISCONNECT
--
--    Disconnect the record stream.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_DISCONNECT is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.DISCONNECT);

--  TASK_RMS_DISPLAY
--
--    Display the file.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_DISPLAY is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.DISPLAY);

--  TASK_RMS_ENTER
--
--    Enter the file.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_ENTER is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.ENTER);

--  TASK_RMS_ERASE
--
--    Erase the file.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_ERASE is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.ERASE);
```

```
--  TASK_RMS_EXTEND
--
--    Extend the file.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_EXTEND is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.EXTEND);

--  TASK_RMS_FIND
--
--    Find a record in the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_FIND is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.FIND);

--  TASK_RMS_FLUSH
--
--    Flush the record.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_FLUSH is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.FLUSH);

--  TASK_RMS_FREE
--
--    Free the record.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_FREE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.FREE);

--  TASK_RMS_GET
--
--    Get a record from the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_GET is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.GET);
```

```
--  TASK_RMS_NXTVOL
--
--    Go to the next volume.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_NXTVOL is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.NXTVOL);

--  TASK_RMS_OPEN
--
--    Open the file.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_OPEN is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.OPEN);

--  TASK_RMS_PARSE
--
--    Parse the file name.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_PARSE is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.PARSE);

--  TASK_RMS_PUT
--
--    Insert a record in the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_PUT is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.PUT);

--  TASK_RMS_READ
--
--    Read a block from the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_READ is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.READ);
```

```
--  TASK_RMS_RELEASE
--
--    Release the record.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_RELEASE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.RELEASE);

--  TASK_RMS_REMOVE
--
--    Remove the file.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_REMOVE is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.REMOVE);

--  TASK_RMS_RENAME
--
--    Rename the file.
--
--    Parameters:
--
--      oldfab      = Address of the old file access block.
--
--      err         = Address of a user error completion routine.
--
--      suc         = Address of a user success completion routine.
--
--      newfab      = Address of the new file access block.
--
procedure TASK_RMS_RENAME is
    new RMS_ASYNCH_2FAB_OPERATION (STARLET.RENAME);

--  TASK_RMS_REWIND
--
--    Rewind the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_REWIND is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.REWIND);
```

```
--  TASK_RMS_SEARCH
--
--    Search for the file name.
--
--    Parameters:
--
--      fab = Address of the file access block.
--
procedure TASK_RMS_SEARCH is
    new RMS_ASYNCH_FAB_OPERATION (STARLET.SEARCH);

--  TASK_RMS_SPACE
--
--    Space to skip in the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_SPACE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.SPACE);

--  TASK_RMS_TRUNCATE
--
--    Truncate the record.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_TRUNCATE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.TRUNCATE);

--  TASK_RMS_UPDATE
--
--    Update the record.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_UPDATE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.UPDATE);
```

```
--  TASK_RMS_WAIT
--
--    Wait for asynchronous record service completion.  This is a
--    renaming of STARLET.WAIT.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_WAIT (
    STATUS   : out COND_VALUE_TYPE;
    RAB      : in out RAB_TYPE) renames STARLET.WAIT;

--  TASK_RMS_WRITE
--
--    Write a block to the file.
--
--    Parameters:
--
--      rab = Address of the record access block.
--
procedure TASK_RMS_WRITE is
    new RMS_ASYNCH_RAB_OPERATION (STARLET.WRITE);

--
-- Additional declarations for compatibility with earlier versions
-- of TASKING_SERVICES.
--
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF   : in  STRING;
    SENDTO   : in  STRING;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  UNSIGNED_LONGWORD := 0;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    IGNORED  : in  ADDRESS           := ADDRESS_ZERO;
    SENDTO   : in  STRING;
    SNDTYP   : in  UNSIGNED_LONGWORD := 0;
    IOSB     : out IO_STATUS_BLOCK_TYPE;
    CARCON   : in  UNSIGNED_LONGWORD := 32;
    FLAGS    : in  UNSIGNED_LONGWORD := 0;
    REQID    : in  UNSIGNED_LONGWORD := 0;
    TIMOUT   : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS   : out COND_VALUE_TYPE;
    EFN      : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF   : in  STRING;
```

```
    IGNORED : in  ADDRESS           := ADDRESS_ZERO;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
    IOSB    : out IO_STATUS_BLOCK_TYPE;
    CARCON  : in  UNSIGNED_LONGWORD := 32;
    FLAGS   : in  UNSIGNED_LONGWORD := 0;
    REQID   : in  UNSIGNED_LONGWORD := 0;
    TIMOUT  : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
    IOSB    : out IO_STATUS_BLOCK_TYPE;
    CARCON  : in  UNSIGNED_LONGWORD := 32;
    FLAGS   : in  UNSIGNED_LONGWORD := 0;
    REQID   : in  UNSIGNED_LONGWORD := 0;
    TIMOUT  : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF  : in  STRING;
    SENDTO  : in  STRING;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
    IOSB    : in  ADDRESS           := ADDRESS_ZERO;
    CARCON  : in  UNSIGNED_LONGWORD := 32;
    FLAGS   : in  UNSIGNED_LONGWORD := 0;
    REQID   : in  UNSIGNED_LONGWORD := 0;
    TIMOUT  : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    IGNORED : in  ADDRESS           := ADDRESS_ZERO;
    SENDTO  : in  STRING;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
    IOSB    : in  ADDRESS           := ADDRESS_ZERO;
    CARCON  : in  UNSIGNED_LONGWORD := 32;
    FLAGS   : in  UNSIGNED_LONGWORD := 0;
    REQID   : in  UNSIGNED_LONGWORD := 0;
    TIMOUT  : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    MSGBUF  : in  STRING;
    IGNORED : in  ADDRESS           := ADDRESS_ZERO;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
    IOSB    : in  ADDRESS           := ADDRESS_ZERO;
    CARCON  : in  UNSIGNED_LONGWORD := 32;
    FLAGS   : in  UNSIGNED_LONGWORD := 0;
    REQID   : in  UNSIGNED_LONGWORD := 0;
    TIMOUT  : in  UNSIGNED_LONGWORD := 0);
procedure TASK_BRKTHRUW (
    STATUS  : out COND_VALUE_TYPE;
    EFN     : in  EF_NUMBER_TYPE    := EF_NUMBER_ZERO;
    SNDTYP  : in  UNSIGNED_LONGWORD := 0;
```

```
        IOSB    : in  ADDRESS           := ADDRESS_ZERO;
        CARCON  : in  UNSIGNED_LONGWORD := 32;
        FLAGS   : in  UNSIGNED_LONGWORD := 0;
        REQID   : in  UNSIGNED_LONGWORD := 0;
        TIMOUT  : in  UNSIGNED_LONGWORD := 0);
    procedure TASK_ENQW (
        STATUS  : out    COND_VALUE_TYPE;
        EFN     : in     EF_NUMBER_TYPE         := EF_NUMBER_ZERO;
        LKMODE  : in     UNSIGNED_LONGWORD;
        LKSB    : in out LOCK_STATUS_BLOCK_TYPE;
        FLAGS   : in     UNSIGNED_LONGWORD      := 0;
        RESNAM  : in     STRING;
        PARID   : in     LOCK_ID_TYPE           := LOCK_ID_ZERO;
        BLKAST  : in     AST_HANDLER            := NO_AST_HANDLER;
        ACMODE  : in     ACCESS_MODE_TYPE       := ACCESS_MODE_ZERO;
        RESERVED: in     ADDRESS                := ADDRESS_ZERO);
    procedure TASK_ENQW (
        STATUS  : out    COND_VALUE_TYPE;
        EFN     : in     EF_NUMBER_TYPE         := EF_NUMBER_ZERO;
        LKMODE  : in     UNSIGNED_LONGWORD;
        LKSB    : in out LOCK_STATUS_BLOCK_TYPE;
        FLAGS   : in     UNSIGNED_LONGWORD      := 0;
        PARID   : in     LOCK_ID_TYPE           := LOCK_ID_ZERO;
        BLKAST  : in     AST_HANDLER            := NO_AST_HANDLER;
        ACMODE  : in     ACCESS_MODE_TYPE       := ACCESS_MODE_ZERO;
        RESERVED: in     ADDRESS                := ADDRESS_ZERO);
private

    -- implementation-defined

end TASKING_SERVICES;
```

Index

A

Abort statement, 8–24
  asynchronous implementation of, 8–25
  synchronous implementation of, 8–25
Access methods
  Ada equivalents for VMS, 6–23
Access modes
  VMS equivalents for VAX Ada, 6–23
Access types
  allocation of collection for, 2–22
  deallocation of storage for, 2–22, 2–43
  effect of length representation clauses on declaration of, 2–22
  packing, 2–25
  passing as parameters by descriptor, 5–31
  passing parameters of, 5–7, 5–20
  representation of, 2–22
  returning as function results, 5–25
  storage size for values of, 2–22
ACCESS_BIT_NAMES_TYPE, B–28
ACCESS_MODE_TYPE, B–28
ACOS, B–26
ACOSD, B–26
Ada
  overview of VAX, 1–1
ADA$INPUT logical name, 3–10, 3–87
ADA$OUTPUT logical name, 3–10, 3–87
ADA$PREDEFINED
  extracting predefined package specifications from, 6–2, B–1
ADA$_EXCCOP condition value
  for marking copied signal arguments in an exception, 4–6
ADA$_EXCCOPLOS condition value
  for marking copied and modified signal arguments in an exception, 4–6
ADA$_EXCEPTION condition value, 4–1, 4–4, 4–6, 4–7, 4–14
ADDRESS attribute, 10–1
  causing locally volatile parameter or variable, 10–1
  effect on storage allocation, 2–42
  using to pass Ada subprograms as parameters, 6–24
Address clauses, 2–24, 2–35
  example of use of, 2–35
$ADDRESS program section, 5–39
ADDRESS type, 2–23, 10–1
Address types
  packing, 2–25
  passing parameters of, 5–20
  representation of, 2–23
  returning as function results, 5–25
Address values
  working with, 10–1
ADDRESS_RANGE_TYPE, B–28
ADDRESS_ZERO
  as default expression for optional parameter, 6–12
ADDRESS_ZERO constant, 6–24
Alignment clauses, 2–33
  restrictions on possible values for, 2–34
Area control block
  using to return array type function results, 5–27
  using to return record type function results, 5–27
Argument list, 5–12
  creation of, 5–9
  passed between languages and system service routines, 6–11

Argument list (cont'd.)
state of optional parameters in calls to system routines, 6–11, 6–12
Argument pointer (AP), 5–14
ARG_LIST_TYPE, B–28
Arrays
assigning values to, 10–10
definition of packable components of, 2–24
example of calculating size of, 2–18
examples of packing, 2–26, 2–27
properties of multidimensional, 2–18
Array types
default alignment of components in, 2–18
effects of packing components of, 2–26
packable, 5–19
packing, 2–25
passing as parameters by descriptor, 5–31
passing parameters of, 5–7, 5–18
representation of, 2–18
representation of multidimensional, 2–18
returning as function results, 5–25
ASIN, B–26
ASIND, B–26
ASSERT package, B–2
ASSERT package instantiation
specification of, B–10
ASSERT_EXCEPTIONS package, B–2
specification of, B–10
ASSERT_GENERIC package, B–2
specification of, B–10
ASTLM (AST Queue Limit) quota
effect of delay statements on, 8–24
AST reentrancy, 8–31
ASTs (Asynchronous System Traps), 8–37, 8–40
constraints on handling, 8–43
delivered to completed or abnormal tasks, 8–43
effect on input-output operations, 3–88
effect on size of task control block, 8–8
examples of handling, 8–45
execution of in tasks, 8–11
handling from tasks, 8–42
rules for Ada routines, 8–44
storage allocated for, 8–43
AST_ENTRY attribute, 8–40
AST_ENTRY pragma, 8–40
effect on size of task control block, 8–9
AST_PACKET_REQUEST_TYPE, B–39
AST_PROCEDURE_TYPE, B–28
Asynchronous input-output, 3–10, 3–67, 3–88
ATAN, B–26
ATAN2, B–26
ATAN2D, B–26
ATAND, B–26
AUX_IO_EXCEPTIONS package, 3–86, B–2
exceptions predefined in, 4–5

B

BASIC
sharing common blocks with, 5–43
Bit array, 5–19
Bit string, 5–18
BLISS
exporting and importing objects from, 5–40
sharing variables with, 5–43
Blocks
as masters of tasks, 8–2
exception handlers for, 4–2
stack frames for, 4–2
BOOLEAN type
packing, 2–24
representation of, 2–2, 2–3
Buffers
control of terminal text file, 3–81
flushing of text file, 3–81
Busy waiting
avoiding during task call to SYS$SETAST, 8–38
avoiding to avoid AST deadlock, 8–44
BYTE_SIGNED_TYPE, B–28
BYTE_UNSIGNED_TYPE, B–28

C

C
sharing variables with, 5–43
CALENDAR package, B–2
CALLABLE attribute
value of during task AST handling, 8–43
Callable utilities
writing interfaces to/from VAX Ada, 6–19
Call-back routines
and generic code sharing, 9–16
example of writing and calling from VAX Ada, 6–36
Call frame, 5–14
CALLG instruction, 5–9, 5–15
CALLS instruction, 5–9, 5–15
Call stack, 5–8
at run time, 5–9
Carriage control
FORTRAN control characters for, 3–84
options for Ada text files, 3–82

Catch-all exception handlers
and fault handlers, 4–25
behavior of, 4–2
CDD (Common Data Dictionary)
examples of using with VAX Ada, 7–5
using with VAX Ada, 7–1
VAX Ada translator utility for, 7–2
CDDL (Common Data Dictionary Language)
VAX Ada equivalent data types for, 7–3
CDD_TYPES package, 7–2, B–3
specification of, B–14
CHANNEL_TYPE, B–28
CHARACTER type
representation of, 2–2, 2–3, 2–25
CHAR_STRING_TYPE, B–28
Checks
method for eliminating run-time, 9–21
suppressing run-time, 4–11
CHF (VAX Condition Handling Facility)
summary of exception-handling implementation, 4–3
used to implement exception handling, 4–1
CLI package, 6–2, B–3
See also System-routine packages
CLOSE procedure
FORM parameter, 3–11
CMS (DEC/Code Management System)
example of calling a routine from Ada, 6–25
CODE, B–19
optional parameter to the pragma IMPORT_EXCEPTION, 4–12
Code Management System (CMS)
See CMS
$CODE program section, 5–38
Collections
allocation of for access types, 2–22
deallocation of for access types, 2–22, 2–43
default allocation for, 2–42
effect of length representation clauses on, 2–22
efficient allocation of, 2–42
Common Data Dictionary
See CDD
Condition codes
See Conditions (VAX)
Condition handlers
calling fault handlers from, 4–25
general VAX Ada, 4–3
Condition handling
See Exception handling
Conditions (VAX)
continuing the signals for from an Ada program, 4–18
effects of handling from an Ada program, 4–21
equivalent Ada predefined exceptions for, 4–7
examples of calling from an Ada program, 4–16
importing into an Ada program, 4–12
matching Ada exceptions with, 4–7
noncontinuable execution of, 4–21
not caught by Ada exception handlers, 4–21
signaling from an Ada program, 4–15
unhandled, 4–2
Condition values
See also Exceptions, Exception handling
giving to Ada exceptions, 4–13
CONDITION_HANDLING package, 6–2, B–3
example of using MATCH_COND function, 6–27
provision of interface to LIB$MATCH_COND, 6–26
specification of, B–19
using to signal VAX conditions from an Ada program, 4–15
using to test status values, 6–26
COND_ID, B–19
COND_VALUE_TYPE, B–19, B–28
$CONSTANT program section, 5–38
Constrainedness bit, 5–20
CONSTRAINT_ERROR
VAX condition equivalent for, 4–8
CONSTRAINT_ERROR exception
checks that raise, 9–21
raised when passing parameters, 5–29, 5–32, 5–33, 5–34
raised when using UNSIGNED_LONGWORD type, 10–8
raised with varying strings, 10–9
underlying run-time checks for, 4–9
VAX condition equivalent for, 4–8
CONTEXT_TYPE, B–28
CONTINUE command (DCL)
entering after CTRL/Y in tasking program, 8–25
Control blocks
declarations of types for in the system-routine packages, 6–7
example of using VMS RMS, 6–34
structure of in the system-routine packages, 6–7
CONTROL_C_INTERCEPTION package, 8–26, B–3
specification of, B–26
Copy-in/copy-back semantics, 5–7
for passing access type parameters, 5–20
for passing address type parameters, 5–20
for passing array type parameters, 5–7

Copy-in/copy-back semantics (cont'd.)
for passing record type parameters, 5–7, 5–20
for passing scalar, access, and address type parameters, 5–7
for passing scalar parameters, 5–18
for passing task type parameters, 5–7, 5–21
COS, B–26
COSD, B–26
COSH, B–26
CPU time
decreasing for a VAX Ada program, 9–20
techniques for reducing, 9–19
CREATE procedure, 3–2, 3–34, 3–35
FILE parameter, 3–6
FORM parameter, 3–3, 3–6
MODE parameter, 3–36
NAME parameter, 3–6
Creation-time attributes
of input-output files, 3–34
CTRL/C
interception with AST entry, 8–46
CTRL/Y
interrupting tasks with, 8–25
CUST_DEF, B–19

D

D_floating representation, 2–5, 2–6, 2–7, 2–8, 2–9, 2–11
D_FLOAT type
representation of, 2–5
storage size of, 2–5
Data
Ada features for optimizing, 2–24
representing in mixed-language programs, 5–16
sharing with non-Ada routines, 5–38
$DATA program section, 5–38
Data structures
VMS, 6–4
DATE_TIME_TYPE, B–28
Deadlock
See Task deadlock
Deallocation
of storage associated with access types, 2–22, 2–43
DEBUG command (DCL)
entering after CTRL/Y in tasking program, 8–25
DEC/CMS
See CMS
DEC Multinational Character Set
See Multinational Character Set
Default parameters
in system routines vs. Ada, 6–11
Delay statement, 8–24
avoiding during task call to SYS$SETAST, 8–37
avoiding to avoid AST deadlock, 8–44
using with abnormal tasks, 8–25
Descriptor classes
explanation of for VAX Ada, 5–31
Descriptor mechanism, 5–13
passing array type parameters by, 5–19
DESCRIPTOR mechanism option
descriptor data types used with, 5–34
for imported function results, 5–36
for imported subprogram parameters, 5–30
type requirements for descriptor classes for, 5–32
valid class names for, 5–32
Descriptors
default used by VAX Ada for passing array type parameters, 5–19
returning function results with, 5–27
using to pass parameters to exported Ada subprograms, 5–37
DESCRIPTOR_TYPE, B–28
DEVICE_NAME_TYPE, B–28
Direct files, 3–4
default attributes for, 3–47
specifying record size for, 3–48
DIRECT_IO
default file attributes provided by, 3–48
DIRECT_IO package, 3–1, 3–4, 3–40, B–3
DIRECT_MIXED_IO package, 3–1, 3–4, 3–41, B–3
default file attributes provided by, 3–48
example of using, 3–49
Discriminants
See Record discriminants
DSC$K_CLASS_A, 5–19, 5–27, 5–37
DSC$K_CLASS_S, 5–37
DSC$K_CLASS_SB, 5–19, 5–27, 5–37
DSC$K_CLASS_UBA, 5–19, 5–27
DSC$K_CLASS_UBS, 5–37
DSC$K_CLASS_UBSB, 5–19, 5–27, 5–37
DTK package, 6–2, B–4
See also System-routine packages
Dynamic memory
use of to allocate storage, 2–42

E

Edit/FDL Utility
using to optimize external files, 3–18
EF_CLUSTER_NAME_TYPE, B–28

EF_NUMBER_TYPE, B–28
ELABORATE pragma
  using to improve run-time performance, 9–22
Elaboration
  order of for programs involving tasks, 8–1
Elapsed time
  decreasing in a VAX Ada program, 9–27
  techniques for reducing, 9–19
END_ERROR
  raised during terminal input-output, 3–69
Entries
  See Task entries
Enumeration clauses, 2–29
  See also Representation clauses
Enumeration types
  declaring signed internal codes for, 2–29
  example of representation of, 2–4
  examples of using representation clauses with, 2–29
  internal codes for literals of, 2–3
  packing, 2–25
  passing parameters of, 5–18
  representation of, 2–3
  returning as function results, 5–25
  specifying internal codes for literals of, 2–29
  storage allocated for objects of, 2–3
Equivalence strings
  for process-permanent files, 3–10
  pairing with logical names, 3–8
Exception handlers
  and VAX conditions, 4–21
  catch-all, 4–2
  general VAX Ada run-time, 4–2
  invoking, 4–3
  search for, 4–2
  unwinding to, 4–2
  VAX Ada run-time, 4–2
  VMS default, 4–2, 4–28
Exception handling, 4–1
  in non-Ada code, 4–7
  making the best use of, 4–8
  relationship to VAX condition handling, 4–1
Exceptions
  Ada format, 4–4, 4–5, 4–6, 4–14
  associating VAX conditions with, 4–13
  avoiding propagation of unhandled, 8–37
  avoiding propagation of unhandled to avoid AST deadlock, 8–44
  copying of signal arguments for, 4–4, 4–6, 4–28
  effect on text file buffers, 3–81
Exceptions (cont'd.)
  example of handling in mixed-language environment, 4–22
  exporting to other languages as VAX conditions, 4–14
  handling in mixed-language programs, 4–11
  importing from other languages, 4–12
  information lost during signal argument copying, 4–6
  input-output, 3–86
  interaction with tasking, 4–27
  matching of imported, 4–7
  matching of user-defined, 4–7
  matching signal arguments of, 4–14
  matching VAX conditions with, 4–7
  mechanism argument vectors for, 4–2
  naming and encoding, 4–5
  noncontinuable execution of, 4–4
  predefined, 4–4, 4–5, 4–7
  propagation of, 4–4, 4–27, 4–28
  raising, 4–2, 4–3
  raising at point of task rendezvous, 4–4
  raising imported, 4–12
  relationship to CHF, 4–3
  re-raising, 4–3, 4–4
  signal argument vectors for, 4–2
  suppressing checks that raise, 4–9, 4–11
  underlying run-time checks for, 4–9
  unhandled in tasking programs, 4–2, 4–27
  user-defined, 4–4, 4–5, 4–6
  VAX condition equivalents for predefined, 4–7
  VAX condition values for predefined, 4–1
  VAX condition values for user-defined, 4–1
  VMS format, 4–4, 4–5, 4–6, 4–14
EXISTENCE_ERROR
  raised when reading Ada relative files, 3–53
EXIT command (DCL)
  entering after CTRL/Y in tasking program, 8–26
Exit handlers
  restrictions on writing in Ada, 8–39
EXIT_HANDLER_BLOCK_TYPE, B–28
EXP, B–26
EXPAND_AST_PACKET_POOL, B–39
Exporting subprograms, 5–4
Export pragmas, 5–4, 5–40
EXPORT_EXCEPTION pragma, 4–11, 4–14
  and NON_ADA_ERROR exception, 4–15
  examples of using, 4–15
  syntax of, 4–14
  using to associate an Ada exception with a VAX condition, 4–14

EXPORT_EXCEPTION pragma (cont'd.)
using to give user-defined exceptions VMS format, 4–6
EXPORT_FUNCTION pragma
syntax for, 5–4
use of in routine interfaces, 6–22
EXPORT_OBJECT pragma
syntax of, 5–40
EXPORT_PROCEDURE pragma
syntax for, 5–4
use of in routine interfaces, 6–22
using in a run-time library routine call, 6–36
EXPORT_VALUED_PROCEDURE pragma
default passing mechanism for first parameter of, 6–23
parameter modes for, 6–22
required mode of first parameter of, 6–23
syntax for, 5–4
treatment of first parameter of, 5–15
use of to write call-back routines, 6–22
External files
See also Files
creation- and run-time attributes of, 3–34
default attributes of, 3–35
naming, 3–6
relationship to file objects, 3–3
specifying attributes of, 3–11

F

F_floating representation, 2–5, 2–6, 2–7, 2–8, 2–9, 2–10
F_FLOAT type
representation of, 2–5
storage size of, 2–5
FAB (file access block)
record type declared for in the package STARLET, 6–7
FAB_TYPE, B–28
FAC_NO, B–19
FAC_SP, B–19
FAO signal arguments
matching of in non-Ada code, 4–14
zeroed during signal argument copying, 4–7
Fault handlers, 4–25
effect of Ada exception handling on, 4–21
method for setting up in VAX Ada, 4–25
restrictions on using in an Ada program, 4–25
FDL (File Definition Language), 3–11
most likely attributes for Ada files, 3–19
primary attributes of, 3–12
FDL (File Definition Language) (cont'd.)
rules for using, 3–17
secondary attributes of, 3–12
using to give values to FORM parameters, 3–11
using to tune external files, 3–33
File objects, 3–2
association with VMS RMS files, 3–3
creating or opening, 3–2
FILE parameter, 3–6
Files
Ada direct, 3–4, 3–11
Ada indexed, 3–5, 3–11
Ada relative, 3–4, 3–11
Ada sequential, 3–4
Ada text, 3–5
buffering text, 3–80
carriage-control attributes for Ada text, 3–82
carriage control of text, 3–81, 3–82
changing creation-time attributes of external, 3–34
consistency checking of attributes of external, 3–35
creation-time attributes of external, 3–34
default attributes for Ada direct, 3–48
default attributes for Ada indexed, 3–55
default attributes for Ada relative, 3–51
default attributes for Ada sequential, 3–45
default attributes for Ada text, 3–65
default attributes for external, 3–35
default characteristics of input-output, 3–3
default logical names for VMS, 3–9
default specifications for, 3–8
defining keys in indexed, 3–5
definition of Ada input-output, 3–3
definition of external, 3–3
external, 3–2
FDL attributes for tuning external, 3–33
FORTRAN carriage-control characters for Ada text, 3–84
input-output, 3–2
locking records in, 3–39
logical names for, 3–8
mixed-type, 3–41
most likely FDL attributes for external, 3–19
naming external, 3–6
optimizing external, 3–18
optimizing performance of, 3–37
process-permanent, 3–10, 3–87
reading indexed, 3–57
run-time attributes of external, 3–34
sharing input-output, 3–36
specifying attributes for external, 3–11

Files (cont'd.)
specifying FDL attributes for external, 3–17
specifying key information for indexed, 3–55
specifying record size for Ada direct, 3–48
specifying record size for Ada relative, 3–51
specifying VMS RMS attributes of, 3–12
terminators in Ada text, 3–77
using FORM parameter to control attributes of external, 3–11
using FORM parameter to control sharing of, 3–36
writing VMS specifications for, 3–7
File specifications
VMS syntax for, 3–7
File terminator
in Ada text file, 3–77
FILE_PROTECTION_FLAGS_TYPE, B–28
FILE_PROTECTION_REC_TYPE, B–28
FILE_PROTECTION_TYPE, B–28
FIRST attribute
using to obtain unsigned numbers, 10–8
Fixed-point type
definition of, 2–16
Fixed-point types
accuracy of, 2–16
packing, 2–25
passing parameters of, 5–18
representation of, 2–16
returning as function results, 5–25
truncation of operations on, 2–16
FLOAT
as parent type for nonpredefined floating-point type, 2–6
Floating-point type
definition of, 2–5
Floating-point types
accuracy of, 2–7
D_floating representation, 2–11
F_floating representation, 2–10
G_floating representation, 2–13
how compiler chooses representation of, 2–7
H_floating representation, 2–14
model numbers defined for, 2–7
packing, 2–25
passing parameters of, 5–18
representation of, 2–5, 2–6
returning as function results, 5–25
safe numbers defined for, 2–9
unchecked conversions to or from, 2–11, 2–12, 2–14, 2–16
See also UNCHECKED_CONVERSION procedure
Floating-point types (cont'd.)
VAX representations and storage sizes for, 2–5
FLOAT type
representation of, 2–5
storage size of, 2–5
FLOAT_MATH_LIB package, A–1
example of using, 6–18
FLOAT_TEXT_IO package, 3–85, A–1
FLOOVEMAT, B–26
FLOUNDMAT, B–26
FORM
See also Exceptions, Ada format
See also Exceptions, VMS format
optional parameter to the pragma IMPORT_EXCEPTION, 4–12
FORM parameter, 3–11, 3–35, 3–36
See also CREATE procedure, OPEN procedure
association with FDL string or file, 3–11
rules for specifying, 3–12
specifying record locking with, 3–39
using to name an external file, 3–6
using to specify carriage-control attributes, 3–81
FORTRAN
exporting an Ada function to, 5–38
handling exceptions propagated from, 4–22
importing a routine from, 5–30
nonreentrancy of run-time library, 8–32
sharing common blocks with, 5–43, 5–44
Frame pointer (FP), 5–14
Frames
call, 5–9, 5–14
definition of Ada versus VMS, 4–3
distinction between Ada and stack, 4–2
exception handlers for, 4–2
Full reentrancy, 8–31
Function results
area control block for returning array type, 5–27
controlling the mechanisms for imported, 5–35
default VAX Ada mechanisms for returning, 5–24
linkage conventions for VAX Ada, 5–16
passing imported by descriptor, 5–36
passing imported by reference, 5–36
passing imported by value, 5–36
registers used for, 5–13
VAX Calling Standard conventions for returning, 5–13
Functions
See Subprograms
FUNCTION_CODE_TYPE, B–28

G

G_floating representation, 2–5, 2–6, 2–8, 2–9, 2–10, 2–13
G_FLOAT type
  representation of, 2–5
  storage size of, 2–5
Garbage collection, 2–43
Generic code sharing
  benefits of, 9–15
  effect on your program, 9–17
  maximizing, 9–16
  performance of code generated for, 9–17
Generic instantiations
  creating library packages of, 9–17
  sharing code for, 9–14
  using to improve program efficiency, 9–17
Generics
  inline expansion of bodies of, 9–13
  making use of, 9–11
  sharing code for, 9–14
  VAX Ada implementation of, 9–12
Generic subprograms
  using the pragma INLINE with instantiations of, 9–5
GET procedure
  default files for TEXT_IO, 3–87
GET_ITEM procedure, 3–41
GET_LINE procedure, 3–69
Global literals
  See Symbol definitions

H

H_floating representation, 2–5, 2–6, 2–7, 2–8, 2–9, 2–14
H_FLOAT type
  representation of, 2–5
  storage size of, 2–5

I

IDENTIFIER_TYPE, B–28
Importing exceptions, 4–12
Importing subprograms, 5–2
Import pragmas, 5–2, 5–40
  See also individual pragmas by name
  using in routine interfaces, 6–19
  using the MECHANISM option for, 5–28
  using to write system- and utility-routine interfaces, 6–2
IMPORT_EXCEPTION pragma, 4–11, 4–12
  and NON_ADA_ERROR exception, 4–15
  examples of using, 4–12
  syntax of, 4–12
  using to associate an Ada exception with a VAX condition, 4–13
  using to give user-defined exceptions VMS format, 4–6
IMPORT_FUNCTION pragma
  syntax for, 5–2
  use of in routine interfaces, 6–22
IMPORT_OBJECT pragma
  syntax of, 5–40
IMPORT_PROCEDURE pragma
  syntax for, 5–2
  use of in routine interfaces, 6–22
IMPORT_VALUED_PROCEDURE pragma
  parameter modes for, 6–22
  required mode of first parameter of, 6–23
  syntax for, 5–2
  treatment of first parameter of, 5–15
  use of in routine interfaces, 6–22
  using to call SYS$TRNLNM system service, 6–43
IMPORT_VALUE function, 6–25
  example of using, 6–46
Indexed files, 3–5
  default attributes for, 3–55
  specifying key information for, 3–55
INDEXED_IO package, 3–1, 3–5, 3–40, B–4
  default file attributes provided by, 3–55
  example of using, 3–58
INDEXED_MIXED_IO package, 3–1, 3–5, 3–41, B–4
  default file attributes provided by, 3–56
  example of using, 3–61
INHIB_MSG, B–19
Inlinable
  definition of, 9–4
Inline expansion
  of generic bodies, 9–12
  of subprograms, 9–3
INLINE pragma, 9–3
  and dependences on generic bodies, 9–6
  examples of, 9–7
  explicit use of, 9–4
  implicit use of, 9–6
  using to improve run-time performance, 9–22
INLINE_GENERIC pragma, 9–11, 9–12
  comparison with the pragma SHARE_GENERIC, 9–12
  effect on compilation unit dependences, 9–13
  examples of, 9–13

INLINE_GENERIC pragma (cont'd.)
syntax of, 9–13
Input
nonterminal, 3–80
terminal, 3–80
Input-output, 3–1
achieving asynchronous, 3–10, 3–67, 3–88
and exception handling, 3–86
and task wait states, 3–87
avoiding during task call to SYS$SETAST, 8–37
avoiding to prevent AST deadlock, 8–44
binary, 3–40
buffering text, 3–80
carriage control in text, 3–81
direct, 3–47
example of using tasks with, 8–2
flushing of buffers at program exit, 8–40
improving, 9–28
indexed, 3–55
interaction of with tasking, 3–86
relative, 3–51
sequential, 3–44
synchronization of operations for, 3–87
terminal, 3–66, 3–69, 3–72, 3–74
text, 3–64
Input-output packages, 3–1
Instantiations
See Generic instantiations
INTEGER type
range of values for, 2–5
representation of, 2–4
storage size of, 2–5
Integer types
declaring unsigned, 2–28
packing, 2–25
passing parameters of, 5–18
range of values for predefined, 2–5
representation of, 2–4
required symmetry of, 10–6
returning as function results, 5–25
INTEGER_TEXT_IO package, 3–85, A–1
INTERFACE pragma
syntax for, 5–2
using in routine interfaces, 6–19, 6–22
Interfaces (routine)
access methods for parameters in, 6–23
default and optional parameters in, 6–24
determining kind of subprogram for, 6–21
determining parameter types for, 6–3
parameter passing mechanisms for, 6–24
writing in VAX Ada, 6–19, 6–20
Interlocked instructions, 2–34
Interlocked queue instructions
example of using, 10–3
operations in the package SYSTEM for, 10–3
INVARGMAT, B–26
IOSB_TYPE, B–28
IO_EXCEPTIONS package, 3–86, B–4
exceptions predefined in, 4–5
IO_STATUS_BLOCK_TYPE, B–28
ITEM_LIST_2_TYPE, B–28
ITEM_LIST_3_TYPE, B–28
ITEM_LIST_PAIR_REC_TYPE, B–28
ITEM_LIST_PAIR_TYPE, B–28
ITEM_QUOTA_REC_TYPE, B–28
ITEM_REC_2_TYPE, B–28
ITEM_REC_TYPE, B–28

J

JSB instruction, 5–15

L

LBR package, 6–2, B–4
See also System-routine packages
Length representation clause
effect on collections allocated for access types, 2–22
Length representation clauses, 2–28
See also Representation clauses
effect of on first named subtypes, 2–3
efficient use of, 2–42
LIB$FILE_SCAN routine
example of calling from Ada, 6–36
LIB$FILE_SCAN_END routine
example of calling from Ada, 6–36
LIB$MATCH_COND routine
interface for in the package CONDITION_HANDLING, 6–26
provided in CONDITION_HANDLING package, 6–2
LIB$SIGNAL routine
provided in CONDITION_HANDLING package, 6–2
use of to implement the raising of exceptions, 4–3
using to signal VAX conditions from an Ada program, 4–15
LIB$STOP routine
provided in CONDITION_HANDLING package, 6–2
used in exception handling, 4–2

LIB$STOP routine (cont'd.)
use of to implement the raising of exceptions, 4–3, 4–4
using to signal VAX conditions from an Ada program, 4–15
LIB package, 6–2, B–4
See also System-routine packages
example of using to call LIB$FILE_SCAN and LIB$FILE_SCAN_END routines, 6–36
Library packages
extracting specifications for VAX Ada predefined, 6–2, B–1
VAX Ada predefined, 9–18, B–1
Line terminator
in Ada text file, 3–77
Linker
program section allocation by, 5–39
using to perform link-time object size checking, 5–42, 5–45
LOCK_ERROR
raised on access to a locked record, 3–39
LOCK_ID_TYPE, B–28
LOCK_STATUS_BLOCK_TYPE, B–28
LOCK_VALUE_BLOCK_TYPE, B–28
LOG, B–26
LOG10, B–26
LOG2, B–26
Logical names, 3–8
using to denote file specifications, 3–8
VMS tables for, 3–9
LOGICAL_NAME_TYPE, B–28
LOGZERNEG, B–26
LONGWORD_SIGNED_TYPE, B–28
LONGWORD_UNSIGNED_TYPE, B–28
LONG_FLOAT
as parent type for nonpredefined floating-point type, 2–6
LONG_FLOAT pragma, 2–9
effect on nonpredefined floating-point types, 2–6
effect on the type LONG_FLOAT, 2–6
using ACS commands to change the value of, 2–10
LONG_FLOAT type
representation of, 2–5
storage size of, 2–5
LONG_FLOAT_MATH_LIB package, A–1
LONG_FLOAT_TEXT_IO package, 3–85, A–1
LONG_LONG_FLOAT
as parent type for nonpredefined floating-point type, 2–6, 2–7
LONG_LONG_FLOAT type
representation of, 2–5
storage size of, 2–5
LONG_LONG_FLOAT_MATH_LIB package, A–1
LONG_LONG_FLOAT_TEXT_IO package, 3–85, A–1
Loop parameters
effect of length representation clauses on, 2–3
Low-level features
using, 10–2

M

MACHINE_SIZE attribute
comparison with SIZE attribute, 2–38
results of for types, 2–39, 2–40
using with types, 2–38
MACRO
exporting and importing objects from, 5–40
sharing variables with, 5–43
Main program
See also Main task
as environment task, 8–1
execution of, 8–1
termination of, 8–2
Main task, 8–1
See also Task stack
controlling size of stack for, 8–13
increasing and decreasing the top guard stack area of, 8–14
increasing and decreasing the working storage area of, 8–14
program region for allocating task stack for, 8–13
size of task control block for, 8–9
MAIN_STORAGE pragma
effect on program region for task stacks, 8–10
to control size and allocation of main task stack, 8–13
using to control size and allocation of main task stack, 8–12
MASK_BYTE_TYPE, B–28
MASK_LONGWORD_TYPE, B–28
MASK_PRIVILEGES_TYPE, B–28
MASK_QUADWORD_TYPE, B–28
MASK_WORD_TYPE, B–28
MATCH_COND, B–19
Math routines
example of importing from VAX Run-Time Library, 4–12
MATH_LIB package, 6–1, 6–17, B–5
predefined instantiations of operations in, 9–18
specification of, B–26

Mechanism arguments
in raising exceptions, 4–2
MECHANISM option, 5–28
Memory
sharing between VAX CPUs, 10–13
Mixed-language programming, 5–1
and data representation, 5–16
conventions for passing data in, 5–6
example of handling exceptions in, 4–22
examples of, 5–3, 5–5
exception handling in, 4–11, 4–21
VAX Calling Standard conventions for, 5–8
with tasks, 8–31
Model numbers
defined for each floating-point type, 2–7
MODE parameter, 3–36
MSG_NO, B–19
MTH package, 6–2, 6–18, B–5
See also System-routine packages
Multinational Character Set
relationship to the type CHARACTER, 2–2, 2–3

N

NAM (name block)
record type declared for in the package STARLET, 6–7
NAME parameter, 3–6
NCS package, 6–2
See also System-routine packages
NEW_LINE, 3–79, 3–80
NEW_PAGE, 3–79, 3–80
NON_ADA_ERROR
as match for imported VAX conditions, 4–7
encoding of, 4–15
NON_ADA_ERROR exception, 4–15
NULL_PARAMETER attribute, 6–12, 6–17, 6–24
example of use in the package STARLET, 6–13

O

Object
definition of an, 2–1
Objects
aligning components of record, 2–33
allocation of storage for, 2–42
controlling stack sizes of task, 8–12
control over representation and storage of, 2–1
declaring for mixed-language programming, 2–34
determining size of, 2–37
dynamic allocation of, 2–42
Objects (cont'd.)
effect of lifetimes on storage allocation, 2–42
how the compiler represents and stores, 2–2
initialization of, 2–35
loop parameter, 2–3
overlaying onto storage locations using address clauses, 2–35
passing to non-Ada routines, 2–29
relationship to types, 2–1
representation and storage of, 2–1
representation and storage of integer, 2–4
representation of, 2–2
representation of address, 2–23
representation of array, 2–18
representation of fixed-point, 2–16
representation of floating-point, 2–5
representation of record, 2–18
representation of task, 2–24
results of size attributes for, 2–40
sharing storage of with non-Ada code, 5–40
size and representation of those designated by access types, 2–22
stack allocation of, 2–42
storage allocated for enumeration, 2–3
storage size of address, 2–23
storage size of task, 2–24
storage sizes of array, 2–18
task, 8–1, 8–2
used in mixed-language programs, 2–2
using SIZE attribute with, 2–37, 2–38
OPEN procedure, 3–2, 3–34, 3–35
FILE parameter, 3–6
FORM parameter, 3–3, 3–6, 3–11, 3–36
MODE parameter, 3–36
NAME parameter, 3–6
Operators
inline expansion of implicit declarations of, 9–5
Optimizations, 9–1
suppressing, 10–2
/OPTIMIZE qualifier (compilation commands)
effect on generics, 9–11
Optional parameters
in system routines versus Ada, 6–11
OTS package, 6–2, B–5
See also System-routine packages
Output
nonterminal, 3–80
terminal, 3–80

P

Packable types, 2–24
Packages
  as masters of tasks, 8–2
  extracting specifications of VAX Ada predefined, 6–2, B–1
  summary of VAX Ada predefined, B–1
  using the VAX Ada system-routine, 6–3
PACK pragma, 2–24
  effect on CHARACTER type, 2–25
  using to change default array representations, 2–18
Page terminator
  in Ada text file, 3–77
PAGE_PROTECTION_TYPE, B–28
Paging
  controlling, 9–28
Parameter passing
  Ada semantics for, 5–7
  between languages and VMS system service routines, 6–11
  in VMS system routines, 6–6
  mechanisms for in system routines, 6–24
  of subprograms in system routines, 6–24
  VAX Ada default mechanisms for, 5–18
  VAX Calling Standard mechanisms for, 5–12
Parameters
  access methods for system or utility routines, 6–23
  Ada semantics for passing, 5–7
  argument list for passing, 5–12
  controlling the passing mechanisms for imported subprogram, 5–28
  default Ada passing mechanisms for, 5–18
  default and optional in system routines, 6–11
  default and optional to callable routines, 6–24
  default descriptor classes for passing array type, 5–19
  default in VAX Ada, 5–6, 6–11
  determining mechanisms for passing, 5–4, 5–6, 5–18
  determining types for in routine interfaces, 6–3
  example of passing by descriptor, 5–31
  mechanisms for passing VAX data type, 5–21
  modes of for imported or exported subprograms, 6–22
  optional in system and run-time library routines, 6–11
  optional in VAX Ada, 5–6
  passing, 5–6
Parameters (cont'd.)
  passing access type, 5–20
  passing Ada subprograms to system routines, 6–24
  passing address type, 5–20
  passing array type, 5–18
  passing by descriptor to exported subprograms, 5–37
  passing imported by descriptor, 5–30
  passing imported by reference, 5–29
  passing imported by value, 5–28
  passing mechanisms for VMS system routines, 6–6
  passing record type, 5–20
  passing scalar type, 5–18
  passing subprograms as, 5–21
  passing task entries as, 5–21
  passing task type, 5–21
  passing to system or utility routines, 6–24
  required modes for in imported and exported subprograms, 6–23
  VAX Calling Standard mechanisms for passing, 5–12
Pascal
  exporting an Ada subprogram to, 5–5
  exporting and importing objects from, 5–40
  sharing variables with, 5–43
Path name
  CDD, 7–2
PCA
  using to improve performance, 9–19
Performance
  See also Run-time performance
  improving CPU, 9–19
  improving run-time, 9–1
Performance and Coverage Analyzer
  See PCA
PL/I
  exporting and importing objects from, 5–40
  sharing variables with, 5–43
PPL package, 6–2, B–5
  See also System-routine packages
Pragmas
  using to control object representation and storage, 2–1
PRIORITY pragma
  for controlling task scheduling, 8–16
  for setting task priorities, 8–16
  using to overcome busy waiting, 8–23
Procedures
  See Subprograms

PROCEDURE_TYPE, B–28
Processor status word (PSW), 5–14
PROCESS_ID_TYPE, B–28
PROCESS_NAME_TYPE, B–28
Program counter (PC), 5–14
Program sections, 5–38
  definition of attributes, 5–39
  establishing with PSECT_OBJECT pragma, 5–43
PROGRAM_ERROR exception
  underlying run-time checks for, 4–9
Psects
  See Program sections
PSECT_OBJECT pragma, 2–2
  attributes of objects specified with, 5–43
  syntax of, 5–43
PUT procedure
  default files for TEXT_IO, 3–87
PUT_ITEM procedure, 3–41
PUT_LINE procedure, 3–80

Q

Queue instructions, 2–34

R

RAB (record access block)
  record type declared for in the package STARLET, 6–7
RAB_TYPE, B–28
Record discriminants
  effect on size of record objects, 2–19
  representation of in record layout, 2–19
Record objects
  controlling allocation of with alignments, 2–34
Record representation clauses, 2–29
  See also Alignment clauses
  See also Representation clauses
  effect on the laying out of records in storage, 2–19
  example of use of, 2–30
  using to conserve space, 2–31
  using to force efficient storage of records, 2–32
Records
  biasing of component values of, 2–32
  definition of packable components of, 2–24
  dynamic components in, 2–20
  efficient storage of, 2–32
  example of calculating storage size of, 2–21
  examples of aligning components of, 2–34
  examples of discriminants in, 2–19
Records (cont'd.)
  examples of using representation clauses with, 2–30
  examples of variant, 2–19
  how the compiler lays out, 2–19
  restrictions on aligning components of, 2–34
  simple, 5–27
Record types
  aligning components of, 2–33
  effects of packing components of, 2–26
  packing, 2–25
  passing parameters of, 5–7, 5–20
  representation clauses with, 2–18, 2–29
  representation of, 2–18
  returning as function results, 5–27
  size of, 2–20
  using representation clauses to force efficient storage of, 2–32
Record variants
  effect of the pragma PACK on, 2–28
  representation clauses with, 2–31
  representation of in record layouts, 2–19
Recursive reentrancy, 8–31
Reentrancy
  avoiding nonreentrancy, 8–32
  definition of kinds of, 8–31
  in mixed-language tasking programs, 8–31
Reference mechanism, 5–13
  passing access type parameters by, 5–20
  passing address type parameters by, 5–20
  passing array type parameters by, 5–19
  passing record type parameters by, 5–20
  passing task type parameters by, 5–21
REFERENCE mechanism option
  for imported function results, 5–36
  for imported subprogram parameters, 5–29
Reference semantics, 5–7
  for passing array type parameters, 5–7, 5–19
  for passing record type parameters, 5–7, 5–20
  for passing task type parameters, 5–7, 5–21
Registers
  defined by the VAX Calling Standard, 5–14
  operations in the package SYSTEM for, 10–3
  used to allocate object storage, 2–42
  used to return function results, 5–13, 5–16
  VAX, 5–14
Relative files, 3–4
  default attrbutes for, 3–51
  specifying record size for, 3–51
RELATIVE_IO package, 3–1, 3–5, B–6
  default attributes provided by, 3–51

RELATIVE_IO package (cont'd.)
example of using, 3–53
RELATIVE_MIXED_IO package, 3–1, 3–5, 3–41, B–6
default file attributes provided by, 3–52
Rendezvous, 8–2
during AST handling, 8–42
tentative, 8–23
Representation clauses, 2–24
effect on result of SIZE attribute, 2–38
enumeration, 2–29
length, 2–28
record, 2–29
specifying alignment with, 2–33
use of to control object representation and storage, 2–1
REQUEST_TIME_SLICE, B–39
RESULT_MECHANISM option, 5–35
RIGHTS_HOLDER_TYPE, B–28
RIGHTS_ID_TYPE, B–28
RMS (Record Management Services)
See also System-routine packages, STARLET package, Interfaces (routine)
calling from an Ada program, 6–1
calling from the package STARLET, 6–13
example of calls to/from the package STARLET, 6–16
example of using control blocks, 6–34
STARLET type declarations for, 6–7
testing condition values returned by, 6–26
RMS services
calling asynchronous from tasks, 8–36
RMS_ASYNCH_OPERATIONS package, B–6
ROPRAND, B–26
Run-time attributes
of input-output files, 3–34
Run-Time Library routines
See VMS Run-Time Library routines
Run-time performance
controlling paging to improve, 9–28
eliminating checks to improve, 9–21
improving, 9–1
overlapping execution to improve, 9–28
reducing subprogram call costs to improve, 9–22
using scalar types and simple operations to improve, 9–25
RU_HANDLE_TYPE, B–28

S

Safe numbers
defined for each floating-point type, 2–9
Scalar types
passing as parameters by descriptor, 5–31
passing parameters of, 5–7, 5–18
returning as function results, 5–25
using to improve run-time performance, 9–25
Scale factor
for fixed-point types, 2–16
SECTION_ACCESS_ID_TYPE, B–28
SECTION_ID_TYPE, B–28
SECTION_NAME_TYPE, B–28
Sequential files, 3–4
default attributes for, 3–44
SEQUENTIAL_IO package, 3–1, 3–4, 3–40, B–6
default file attributes provided by, 3–45
example of using, 3–46
SEQUENTIAL_MIXED_IO package, 3–1, 3–4, 3–41, B–6
default file attributes provided by, 3–45
Serial reentrancy, 8–31
SEVERITY, B–19
Shared data
in mixed-language programs, 5–38
Shared memory
example of between VAX CPUs, 10–13
SHARED pragma, 8–27
comparison with the pragma VOLATILE, 8–29
effect of, 8–28
Shared variables
in tasking program, 8–27
SHARE_GENERIC pragma, 9–11, 9–14
comparison with the pragma INLINE_GENERIC, 9–12
examples of using, 9–14
syntax of, 9–14
SHORT_INTEGER type
range of values for, 2–5
representation of, 2–4
storage size of, 2–5
SHORT_INTEGER_TEXT_IO package, 3–85, A–1
SHORT_SHORT_INTEGER type
range of values for, 2–5
representation of, 2–4
storage size of, 2–5
SHORT_SHORT_INTEGER_TEXT_IO package, 3–85, A–1
SIGNAL, B–19
Signal arguments
copying of during exception handling, 4–6, 4–28
information lost during exception handling, 4–6
in raising exceptions, 4–2

Signal arguments (cont'd.)
matching in mixed-language exception handling, 4–14
Simple record, 5–27
SIN, B–26
SIND, B–26
SINH, B–26
SIZE attribute
comparison of results of for types and objects, 2–39, 2–40
comparison with MACHINE_SIZE attribute, 2–38
using to determine the size of objects and types, 2–37
SKIP_LINE procedure, 3–69
SMG package, 6–2, B–7
See also System-routine packages
SOR package, 6–2, B–7
See also System-routine packages
SQRT, B–26
SQUROONEG, B–26
SS$_ACCVIO violation
occurrence of in mixed-language tasking programs, 8–15
occurrence of in tasking programs, 8–11
SS$_DEBUG condition
and Ada exception handlers, 4–21
SS$_FLTDIV condition
VAX Ada equivalent for, 4–8
SS$_FLTDIV_F condition
VAX Ada equivalent for, 4–8
SS$_FLTOVF condition
VAX Ada equivalent for, 4–8
SS$_FLTOVF_F condition
VAX Ada equivalent for, 4–8
SS$_INTDIV condition
VAX Ada equivalent for, 4–8
SS$_INTOVF condition
VAX Ada equivalent for, 4–8
SS$_UNWIND condition
and Ada exception handlers, 4–21
STANDARD package, B–7
exceptions predefined in, 4–5
recompilation of with the pragma LONG_FLOAT, 2–9
STARLET package, 6–1, 8–35, B–7
See also System-routine packages
example of using to call SYS$GETQUI system service, 6–31
example of using to call SYS$TRNLNM system service, 6–29
STARLET package (cont'd.)
example of using VMS RMS control blocks from, 6–34
obtaining specifications for types and operations in, 6–2
severity codes provided in, 6–26
specification of types defined in, B–28
type declarations in for VMS RMS control blocks, 6–7
use of the pragma IMPORT_VALUED_PROCEDURE in, 6–22
use of underscores in routine names in, 6–7
Static memory
use of to allocate storage, 2–42
Status values
constants defined in the package STARLET, 6–27
STOP, B–19
STOP command (DCL)
entering after CTRL/Y in tasking program, 8–25
Storage
controlling for programs with tasks, 8–9
Storage allocation, 2–41, 2–42
for tasks, 8–9
improving efficiency of, 2–42
Storage deallocation, 2–41, 2–43
for tasks, 8–9
STORAGE_ERROR exception
not being raised in mixed-language programs, 8–14
raising of for task stack overflow, 8–11, 8–14
raising of in tasking programs, 8–9
underlying run-time checks for, 4–9
STORAGE_SIZE attribute
application of to tasks, 8–12
using to control size and allocation of task stacks, 8–12
String
definition of, 5–18
Strings
working with varying, 10–9
STRING type
packability of, 2–25
representation of, 2–2
String types
passing as parameters with DESCRIPTOR mechanism option, 5–30
returning as function results, 5–25
STR package, 6–2
See also System-routine packages
Subprograms
Ada semantics for calling, 5–7

Subprograms (cont'd.)
as masters of tasks, 8–2
calling Ada from external routines, 5–4
calling external routines from Ada, 5–2
calling from non-Ada AST service routines, 8–43
controlling the parameter-passing mechanisms for imported, 5–28
effect of implicit inline expansion on, 9–7
effect of the pragma INLINE on library, 9–9
examples of inline expansion of, 9–7
explicit inline expansion of, 9–4
implicit inline expansion of, 9–6
inline expansion of, 9–3
inline expansion of calls to, 9–6
inline expansion of derived, 9–5
inline expansion of generic instantiations of, 9–5
inline expansion of specifications and bodies, 9–7
passing as parameters, 5–21
passing as parameters to system routines, 6–24
reducing costs for calls of, 9–22
special effects of the pragma INLINE on generic, 9–9
use of in routine interfaces, 6–21
VAX Ada linkage conventions for calls to, 5–15
SUCCESS, B–19
SUPPRESS pragma
using to suppress run-time checks, 4–11
SUPPRESS_ALL pragma
using to suppress run-time checks, 4–11, 9–21
Symbol definitions
obtaining, 6–25
SYS$ASSIGN system service
example of specification and calls to, 6–13
SYS$COMMAND logical name, 3–9, 3–87
equivalence strings for, 3–11
representing process-permanent file, 3–10
SYS$CRMPSC system service
example of using, 6–34
SYS$DCLEXH system service
calling from tasks, 8–39
SYS$DEQ system service
example of specification and calls to, 6–14
SYS$DISK logical name, 3–9
SYS$ERROR logical name, 3–9, 3–87
equivalence strings for, 3–11
output file for error messages, 4–3
representing process-permanent file, 3–10
SYS$EXIT system service
avoiding calls to/from tasks, 8–38
SYS$GETQUI system service
calling using the package STARLET, 6–31
SYS$HIBER system service
avoiding calls to/from tasks, 8–38
SYS$INPUT logical name, 3–9, 3–87
equivalence strings for, 3–11
representing process-permanent file, 3–10
SYS$LOGIN logical name, 3–9
SYS$NET logical name, 3–10
SYS$OPEN RMS routine
example of calling, 6–34
SYS$OUTPUT logical name, 3–10, 3–87
equivalence strings for, 3–11
output file for error messages, 4–3
representing process-permanent file, 3–10
SYS$SCRATCH logical name, 3–10
SYS$SETAST system service, 8–37
SYS$SETIMR routine
use of to implement delay statements, 8–24
SYS$TRNLNM
example of Ada routine interface for, 6–20
SYS$TRNLNM system service
calling using the package STARLET, 6–29
example of calling using the pragma IMPORT_VALUED_PROCEDURE, 6–43
SYS$UNWIND system service
use of to invoke an exception handler, 4–3
SYS$WRITE RMS routine
example of specification and calls to, 6–16
SYSTEM package, 6–1, B–8
exceptions predefined in, 4–5
NON_ADA_ERROR in, 4–7
type ADDRESS in, 2–23, 10–1
unsigned types in, 10–6
using types and operations declared in, 10–2
System-routine packages
See also individual packages by name
default and optional parameters in, 6–11
examples of using, 6–29
naming conventions in, 6–7
obtaining specifications for, 6–2, B–1
parameter-passing mechanisms in, 6–6
parameter types used in, 6–3
provision of initialization constants for record types, 6–9
record type declarations in, 6–7
reserved fields in record components in, 6–10
rules for default and optional parameters, 6–12
steps for calling routines with optional parameters, 6–13
System routines
declaring record types for, 2–29
writing interfaces to/from VAX Ada, 6–19

System services
See VMS system services
SYSTEM_RUNTIME_TUNING package, 6–2, B–8
specification of, B–39
using in programs that call asynchronous system services, 6–17

T

TAN, B–26
TAND, B–26
TANH, B–26
Task
definition of, 8–1
Task control block, 8–2, 8–8
address of as value of task object, 2–24
estimating size of, 8–8
Task deadlock, 8–17
caused by SYS$SETAST, 8–37
circular-calling, 8–20
due to busy waiting, 8–23
during AST handling, 8–42
during call from non-Ada AST service routine, 8–43
dynamic-circular-calling, 8–21
exception-induced, 8–18
self-calling, 8–18
Task entries
passing as parameters, 5–21
Tasking
interaction of with input-output, 3–86
TASKING_SERVICES package, 6–1, 8–35, 8–36, 8–37, 8–40, B–8
specification of, B–42
use of overloading for optional parameters in, 6–17
Tasks, 8–1
See also Main task
as masters of tasks, 8–2
busy waiting of, 8–22, 8–24
calling non-Ada routines from, 8–31
calling system services from, 8–35, 8–37
changing priority to improve performance, 8–49
controlling stack sizes of, 8–12
coordination of information among, 8–30
deadlock with, 8–17
definition of suspension of, 8–15
delivery of ASTs to completed or abnormal, 8–43
dependence on masters, 8–2, 8–8
effect of priority on action taken after CTRL/Y, 8–26
Tasks (cont'd.)
effects of system service calls on, 8–35
environment, 8–1
example of serializing, 8–34
example of using, 8–2
first-in-first-out scheduling of, 8–15
handling ASTs from, 8–42
improving run-time behavior with, 9–28
in context of single process, 8–2
increasing concurrency of when calling system services, 8–36
increasing concurrency with TASKING_SERVICES, 8–36
interaction with exception handling, 4–27
interference of busy waiting with scheduling, 8–23
interrupting with CTRL/Y, 8–25
main, 8–1, 8–9
measuring and tuning performance, 8–48
preventing termination messages from, 4–27
priorities and responsiveness of, 8–16
raising exceptions at point of rendezvous, 4–4
reentrancy with, 8–31
round-robin scheduling of, 8–16
scheduling of, 8–15
scheduling of during system service calls, 8–36
serializing to prevent reentry, 8–35
sharing variables with, 8–27
special considerations in using, 8–17
storage allocated for, 8–7
storage allocated for when AST delivered, 8–43
switching of, 8–15
synchronization of input-output operations in, 3–87
system services to avoid calling from, 8–37
tentative rendezvous with, 8–23
termination messages for, 4–28
termination of, 4–28, 8–2, 8–24
using abort statements in, 8–24
using delay statements in, 8–24
using delay statement to force completion of abnormal, 8–25
VAX Ada scheduling strategy for, 8–15
wait states caused by input-output operations, 3–87
Task scheduling, 8–15
during system service calls, 8–36
round-robin, 8–16
Task stack, 8–2, 8–9
See also Main task
default top guard area of, 8–11
default working area of, 8–11
detecting overflow of, 8–11

Task stack (cont'd.)
fixed-size, 8–10, 8–11
for main task, 8–10
increasing and decreasing the top guard area of, 8–11
increasing and decreasing the working area of, 8–11
limits of, 8–10
overflow when calling non-Ada code, 8–14
program region for allocating main, 8–13
reasons for specifying size of, 8–11
top guard area of, 8–10
using top guard area for detecting overflow, 8–15
working storage area of, 8–10
Task switching, 8–2, 8–15
Task synchronization, 8–2
Task types
packing, 2–25
passing parameters of, 5–7, 5–21
representation of, 2–24
returning as function results, 5–25
TASK_BRKTHRUW, B–42
TASK_ENQW, B–42
TASK_GETDVIW, B–42
TASK_GETJPIW, B–42
TASK_GETLKIW, B–42
TASK_GETQUIW, B–42
TASK_GETSYIW, B–42
TASK_QIOW, B–42
TASK_RMS_CLOSE, B–42
TASK_RMS_CONNECT, B–42
TASK_RMS_CREATE, B–42
TASK_RMS_DELETE, B–42
TASK_RMS_DISCONNECT, B–42
TASK_RMS_DISPLAY, B–42
TASK_RMS_ENTER, B–42
TASK_RMS_ERASE, B–42
TASK_RMS_EXTEND, B–42
TASK_RMS_FIND, B–42
TASK_RMS_FLUSH, B–42
TASK_RMS_FREE, B–42
TASK_RMS_GET, B–42
TASK_RMS_NXTVOL, B–42
TASK_RMS_OPEN, B–42
TASK_RMS_PARSE, B–42
TASK_RMS_PUT, B–42
TASK_RMS_READ, B–42
TASK_RMS_RELEASE, B–42
TASK_RMS_REMOVE, B–42
TASK_RMS_RENAME, B–42
TASK_RMS_REWIND, B–42
TASK_RMS_SEARCH, B–42
TASK_RMS_SPACE, B–42
TASK_RMS_TRUNCATE, B–42
TASK_RMS_UPDATE, B–42
TASK_RMS_WAIT, B–42
TASK_RMS_WRITE, B–42
TASK_SNDJBCW, B–42
TASK_STORAGE pragma
application of to tasks, 8–12
to control task stack top guard area, 8–12
using to control size of task stacks, 8–12
TASK_UPDSECW, B–42
Terminal input-output, 3–66
achieving asynchronous, 3–10, 3–67, 3–88
buffering, 3–80
data-oriented method for, 3–72
flexible method for, 3–74
line-oriented method for, 3–69
mixed method for, 3–74
TERMINATED attribute
value of during task AST handling, 8–43
Text files, 3–5
carriage control in, 3–81
default attributes for, 3–65
terminators in, 3–77
VAX Ada implementation of, 3–66
TEXT_IO
predefined instantiations of packages in, 3–85
TEXT_IO package, 3–1, 3–6, 3–64, B–9
carriage control in, 3–81
default file attributes provided by, 3–65
default files for GET procedures, 3–87
default files for PUT procedures, 3–87
example of using, 3–67
predefined instantiations of operations in, 9–18
using for terminal input-output, 3–66
TIME_NAME_TYPE, B–28
TIME_SLICE pragma, 8–49
effect on TQELM quota, 8–24
See also TQELM
recommended values for, 8–17
using to cause round-robin task scheduling, 8–16
using to control task scheduling, 8–16
using to overcome busy waiting, 8–23
TQELM (Timer Queue Entry Limit) quota
effect on delay statements, 8–24
TRANSACTION_ID_TYPE, B–28
TT logical name, 3–10, 3–88
Types
See also individual types by name
Ada equivalents for VAX data, 5–21

Types (cont'd.)
determining size of, 2–37
packable, 2–24
relationship to objects, 2–1
representation of, 2–2
results of size attributes for, 2–40
task, 8–1
unsigned, in the package SYSTEM, 10–6
using MACHINE_SIZE attribute with, 2–38
using SIZE attribute with, 2–37
VAX Ada equivalents for CDD, 7–3

U

UIC_LONGWORD_TYPE, B–28
UIC_TYPE, B–28
Unchecked conversions
See also UNCHECKED_CONVERSION procedure
between address and access types, 2–22
to floating-point types, 2–11, 2–12, 2–14, 2–16
Unchecked deallocation
using to control access type storage, 2–43
UNCHECKED_CONVERSION procedure
between address and access types, 2–22
effect of the pragma INLINE on an instantiation of, 9–5
UNCHECKED_DEALLOCATION procedure, 2–43
example of using, 2–43
using to control access type storage, 2–43
Unsigned types
explanation of Ada, 10–6
in the package SYSTEM, 10–6
UNSIGNED_BYTE type, 10–6
UNSIGNED_LONGWORD type, 10–6
characteristics of, 10–6
UNSIGNED_WORD type, 10–6
Usages (VMS)
See VMS data structures
USER_ARG_TYPE, B–28
USE_ERROR exception
raised for concurrent opening of magnetic tape files, 3–37
raised for FDL errors in FORM parameter, 3–17
raised for mismatch of file attributes, 3–34, 3–35
raised for opening an open file, 3–37
raised on access to a locked record, 3–39
raised when writing text files, 3–66
Utility routines
See VMS utility routines

V

Value mechanism, 5–12
VALUE mechanism option
for imported function results, 5–36
for imported subprogram parameters, 5–28
Variables
effect of the pragma SHARED on, 8–28
effect of the pragma VOLATILE on, 8–29
Varying strings
declaring in VAX Ada, 5–16
working with, 10–9
VAX Calling Standard, 5–8
VAX Common Language Environment, 2–2
VAX data types
VAX Ada equivalents for, 5–21
VAX Performance and Coverage Analyzer
See PCA
VAX Procedure Calling and Condition Handling Standard
conformance of Run-Time Library routines to, 6–11
conformance of system services to, 6–11
conformance of VAX Ada to, 5–1
VAX Procedure Calling Standard
interface for function return values, 5–13
VECTOR_BYTE_SIGNED_TYPE, B–28
VECTOR_BYTE_UNSIGNED_TYPE, B–28
VECTOR_LONGWORD_SIGNED_TYPE, B–28
VECTOR_LONGWORD_UNSIGNED_TYPE, B–28
VECTOR_WORD_SIGNED_TYPE, B–28
VECTOR_WORD_UNSIGNED_TYPE, B–28
VMS data structures, 6–4
VMS Linker
See Linker
VMS Run-Time Library routines
See also System-routine packages, individual packages by name, Interfaces (routine)
calling from an Ada program, 6–1
calling from tasks, 8–11
calling mathematical from VAX Ada, 6–17
declaring record types for, 2–29
example of calling, 6–36
testing condition values returned by, 6–26
VMS system routines
See System routines
VMS system services
See also System-routine packages, STARLET package, Interfaces (routine)
calling asynchronous, 6–17
calling asynchronous from tasks, 8–36

VMS system services (cont'd.)
calling from an Ada program, 6–1
calling from tasks, 8–11
calling from the package STARLET, 6–13
example of calling using the package STARLET, 6–29, 6–31
example of item-list structure in call to, 6–29, 6–31
examples of calls to/from the package STARLET, 6–13
testing condition values returned by, 6–26
VOLATILE pragma, 8–27
comparison with the pragma SHARED, 8–29
effect of, 8–29
effect on storage allocation, 2–42
example of use in system service call, 6–29
example of using with VMS RMS control blocks, 6–34
using with address objects, 10–1
with address determined by ADDRESS attribute, 10–1

W

WORD_COND_VALUE_TYPE, B–19

X

XAB (extended attribute block)
record type declared for in the package STARLET, 6–7

Z

$ZERO program section, 5–39

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local DIGITAL subsidiary or approved distributor
Internal[1]	————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX Ada Run-Time Reference Manual
AA–EF88B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Reader's Comments

VAX Ada Run-Time Reference Manual
AA–EF88B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------