DEC Ada

Run-Time Reference Manual for OpenVMS Systems

Order Number: AA–PWGZB–TK

February 1995

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

Revision/Update Information:	This revised manual supersedes the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* (Order No.: AA–PWGZA–TE).
Operating System and Version:	VMS VAX Version 5.4 or higher OpenVMS Alpha Version 6.1 or higher
Software Version:	DEC Ada Version 3.2

Digital Equipment Corporation
Maynard, Massachusetts

February 1995
Revised, May 1989
Revised, January 1993
Revised, February 1995

ZK5575

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

2 Input-Output Facilities

3 Exception Handling

4 Mixed-Language Programming

5 Calling System or Other Callable Routines

6 Using CDD/Repository from DEC Ada

7 Tasking

8 Improving Run-Time Performance

9 Additional Programming Considerations

A DEC Ada Predefined Instantiations

B Implementation Details Related to Mixed-Language Programs on OpenVMS Systems

Examples

Figures

Tables

Preface

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 *DEC Ada Language Reference Manual*.

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

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

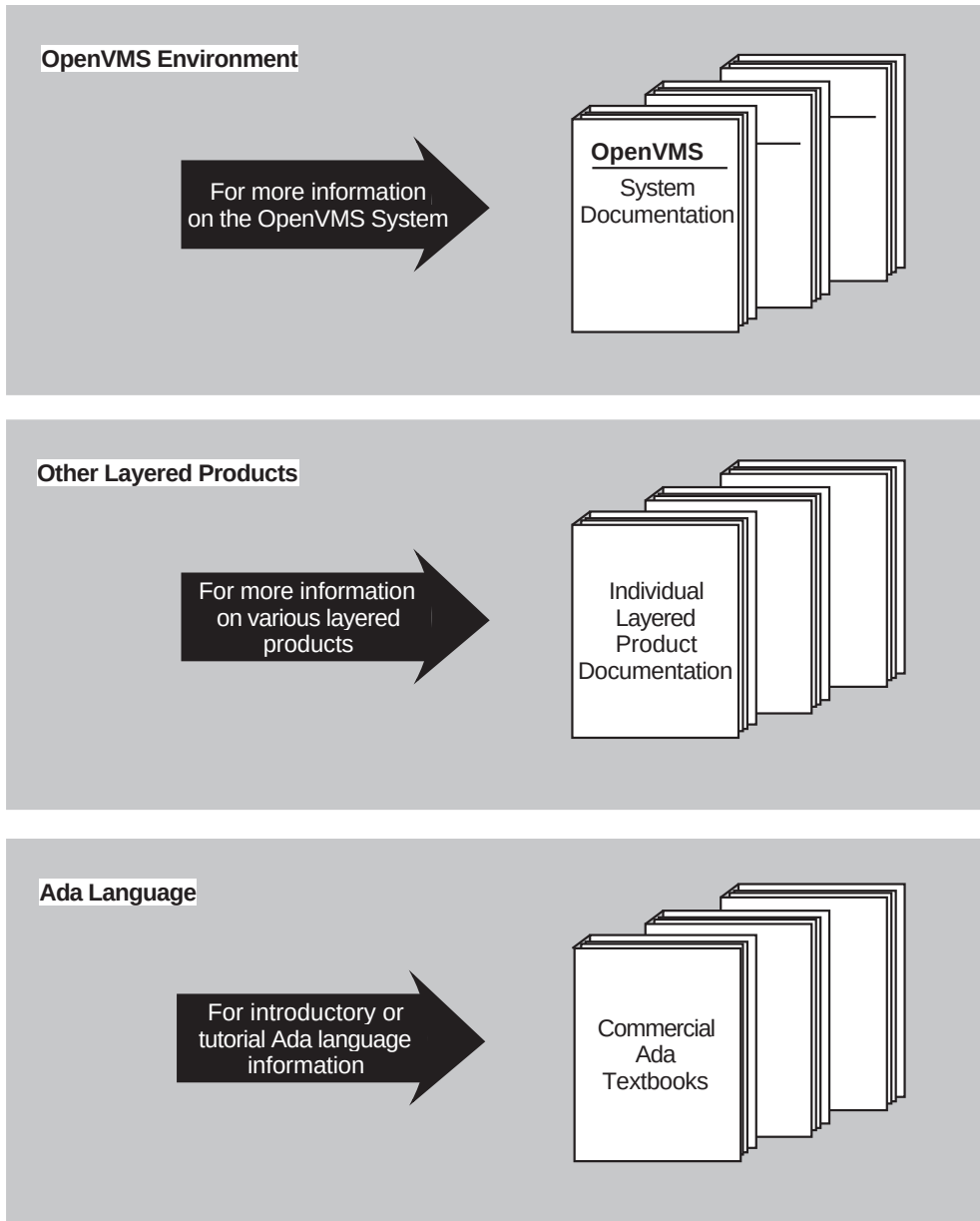
Intended Audience

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 programming language. The reader should have a working knowledge of the Ada language and some familiarity with the operating system.

Documentation Reading Path

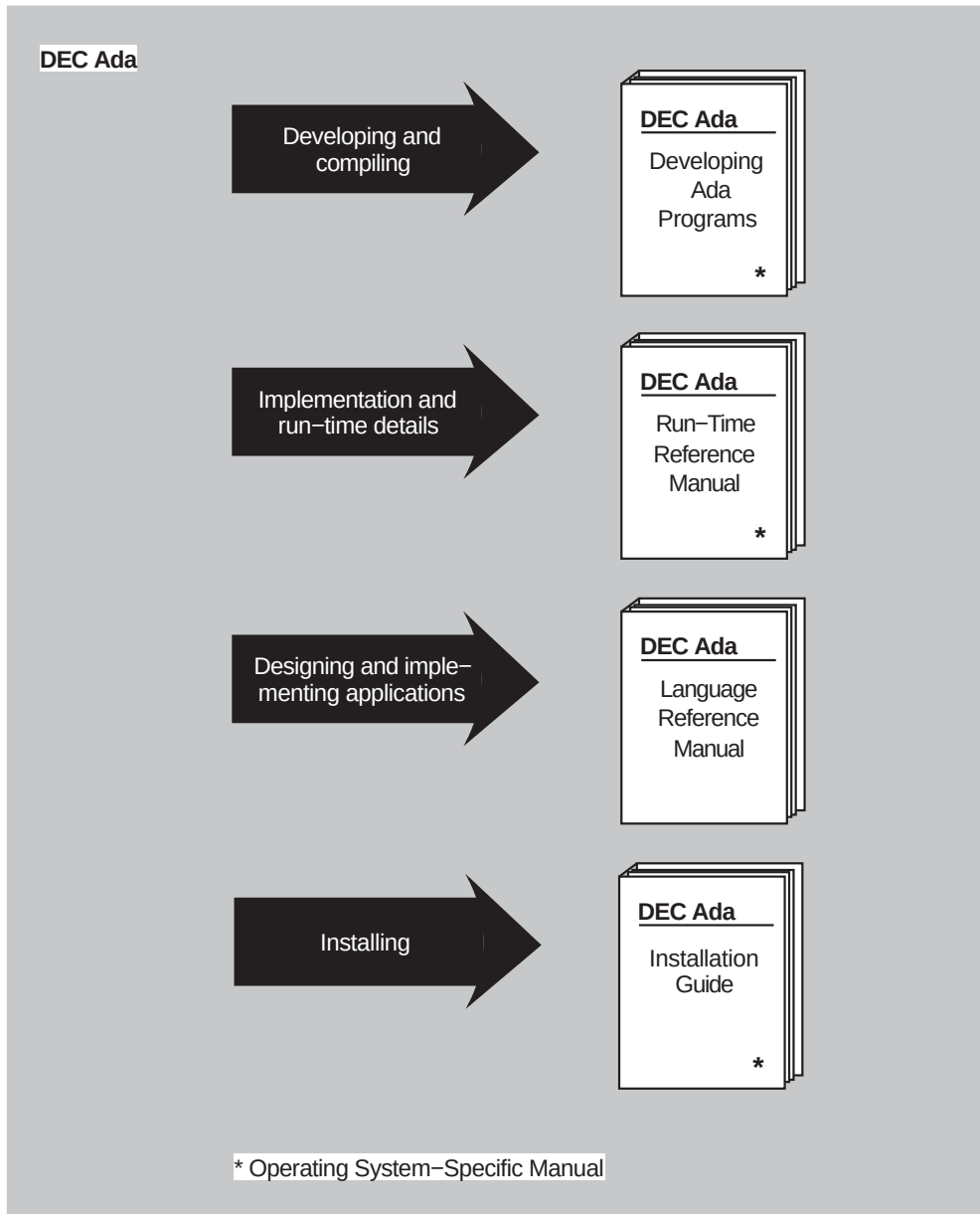
Figures 1 and 2 show the relationship of the Ada documentation set to other documentation that may be helpful.

Figure 1 Documentation Reading Path for Related Documents



ZK–5349A–2–GE

Figure 2 Documentation Reading Path for DEC Ada Documentation



ZK−5349A−1−GE

Document Structure

This manual contains the following chapters and appendixes:

- Chapter 1 explains how DEC Ada objects and types are represented and sized. It also gives information on sharing object storage among Ada and non-Ada routines.
- Chapter 2 discusses DEC Ada input-output, giving details about file sharing, record locking, and the DEC Ada input-output packages. This chapter also summarizes information about the OpenVMS File Definition Language and the specification of file names.
- Chapter 3 describes the implementation of DEC Ada exception handling and discusses the importing and exporting of OpenVMS conditions and Ada exceptions.
- Chapter 4 describes the DEC Ada parameter-passing mechanisms and import-export pragmas and discusses how to write mixed-language programs that involve DEC Ada.
- Chapter 5 explains how to call system and other callable routines (OpenVMS system services, Run-Time Library routines, and so on).
- Chapter 6 describes how to access CDD/Repository FROM DEC Ada.
- Chapter 7 discusses tasking issues, including issues related to calling non-Ada routines (such as OpenVMS system services) from tasks.
- Chapter 8 gives information on how to make DEC Ada programs more efficient.
- Chapter 9 discusses additional details of DEC Ada that you need to consider when writing DEC Ada programs.
- Appendix A lists all of the DEC Ada predefined generic instantiations.
- Appendix B provides implementation details related to mixed-language programming.
- Appendix C lists all of the DEC Ada packages.

Conventions

Table 1 shows the conventions used in this manual.

Table 1 Conventions Used in This Manual

Convention	Description
VMS systems	Refers to OpenVMS Alpha and OpenVMS VAX systems unless otherwise specified.
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
Return	In interactive examples, a label enclosed in a box indicates that you press a key on the terminal, for example, Return.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
`boldface monospace text`	In interactive examples, boldface monospace text represents user input.
file-spec, . . .	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates that additional parameters, options, or values can be entered.
n	A lowercase italic *n* indicates the generic use of a number.
. . .	A horizontal ellipsis in an Ada example or figure indicates that not all of the statements are shown.
. . .	A vertical ellipsis in an interactive figure or example indicates that not all of the commands and responses are shown.
( )	In format descriptions, if you choose more than one option, parentheses indicate that you must enclose the choices in parentheses.
[expression]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{, mechanism_name }	Braces in Ada syntax indicate that the enclosed item can be repeated zero or more times. Braces in debugger command syntax enclose lists from which you must choose one item.
boldface text	Boldface text indicates Ada reserved words.

(continued on next page)

Table 1 (Cont.) Conventions Used in This Manual

Convention	Description
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*.)
*type*_name	Italicized words in syntax descriptions indicate descriptive prefixes that are intended to give additional semantic information rather than to define a separate syntactic catgegory.
UPPERCASE TEXT	Uppercase indicates the name of a command, routine, parameter, procedure, utility, file, file protection code, or the abbreviation for a system privilege.

New and Changed Features

This release improves the functionality of DEC Ada.

The following lists changes for the DEC Ada Version 3.2 user:

- Changes have been made to the packages SYSTEM.
- Support included for passive tasks and pragma PASSIVE, which can significantly improve the performance of rendezvous in programs. Requirements for passive tasks are detailed Chapter 7.
- Support included for the Professional Development Option on both the OpenVMS Alpha and OpenVMS VAX platforms.
- Support for 64-bit integers and floating point numbers has been added to the OpenVMS Alpha platform.
- The implementation of AI–00866, which permits an 8-bit character set based on ISO standard 8859/1 (commonly known as Latin–1) has been added.

In addition to the information in this manual, see the release notes and the *DEC Ada Language Reference Manual* for more information about these changes.

1

Object Representation and Storage

An Ada *object* is an entity that can have values of a particular type. For each Ada object, the DEC 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, the amount of memory required for the generated code, and the amount of memory required for the objects.

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 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 or exported

To increase efficiency, the DEC Ada compiler may use alternative representations for some objects. However, the compiler does not choose an alternative representation for objects that are visible outside the Ada program. It does not choose an alternative representation for any of the objects in the previous list.

This chapter discusses the following topics:

- The representation and storage chosen by the DEC Ada compiler for objects of a variety of DEC Ada types (Section 1.1)
- How to tailor the representation of the objects in your program to suit your particular application (Section 1.2)
- Methods for determining how much storage has been allocated for particular types and objects
- Storage allocation and deallocation (Section 1.4)

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

1.1 Type and Object Representations

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

1.1.1 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 .. 255[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 *DEC Ada Language Reference Manual*.

Section 1.2.4 explains how to use an enumeration representation clause to specify other values (including negative values) for internal codes.

DEC Ada implements AI-00866, which permits an 8-bit character set based on ISO standard 8859/1 (commonly known as Latin-1).

Changes to the definition of the enumeration type CHARACTER, which are permitted by AI-00866, cause some previously correct DEC Ada programs to be in error. For example:

- Programs that assume that the representation of CHARACTER'LAST is 127 or that type CHARACTER has 128 values need to be changed. The representation of CHARACTER'LAST is 255, and type CHARACTER has 256 values.
- Programs that assume that the DEL character is CHARACTER'LAST need to be changed.
- Programs that assume that the representations of all upper- or lower- case characters are contiguous are incorrect and should be changed.
- Programs that include a case statement with an expression of the type CHARACTER (or a type derived from the type CHARACTER) may require modified choices or additional case statement alternatives.
- Programs that include a record variant part with a discriminant of type CHARACTER (or a type derived from the type CHARACTER) may require modified choices or additional variants.
- Programs that include a record representation clause that assume that the type CHARACTER is 7 bits long are incorrect.
- Programs that count all the elements of the type CHARACTER using an 8-bit integer type, such as SHORT_SHORT_INTEGER, may be incorrect. The value of SHORT_SHORT_INTEGER'LAST is 127, and the representation of CHARACTER'LAST is 255.

The following are restrictions on the use of 8-bit characters:

- The debugger does not support extended characters in identifiers.
- Some devices (some printers, for example) do not display all graphic characters in the Latin-1 character set.
- Latin-1 differs slightly from the DEC 8-bit multinational character set. You should set your terminal or window to ISO Latin-1 mode to correctly display Latin-1 characters.

The amount of storage that the DEC Ada allocates 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 *DEC Ada Language Reference Manual* for more information.)

When you specify a length representation clause for a first named subtype, the clause can not be applied to the representation of objects of the base type. For example, this effect may occur with loop parameters.

For simple enumeration objects and enumeration components of unpacked arrays and records, the DEC Ada compiler chooses 1 byte (8 bits), 2 bytes (16 bits), 4 bytes (32 bits), or 8 bytes (64 bits on Alpha systems)—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 it 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, use an enumeration representation clause (see Section 1.2.4).

1.1.2 Integer Types and Objects

DEC Ada provides four predefined integer types:

SHORT_SHORT_INTEGER
SHORT_INTEGER
INTEGER
LONG_INTEGER

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

Values for objects of all four 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 1.2.3). However, because of the way the Ada language defines integer operations, operations on these unsigned objects will involve signed intermediate values.

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

Table 1–1 Range of Values and Storage Sizes for DEC 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} .. $2^{15} - 1$	16
INTEGER	-2^{31} .. $2^{31} - 1$	32
LONG_INTEGER	-2^{31} .. $2^{31} - 1$	32[1]

[1]In Alpha, the value of LONG_INTEGER is 64 bits. User-defined integer types can now be 64 bits, provided that their range is large enough.

See Chapter 9 for more information on working with unsigned types.

1.1.3 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 DEC Ada compiler chooses a hardware floating-point representation, depending on:

- The required range and accuracy of the DEC Ada pragma FLOAT_REPRESENTATION
- The value of the DEC Ada pragma FLOAT_REPRESENTATION

On VAX systems, the following floating-point representations are possible:

F_floating
D_floating
G_floating
H_floating

On Alpha systems, the following representations are possible:

F_floating
D_floating
G_floating
IEEE single float
IEEE double float

The compiler uses chosen representation and size 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.

For detailed information about the layout of VAX and Alpha floating-point representations, see the following manuals:

- *VAX Architecture Handbook*
- *VAX Architecture Reference Manual*
- *Alpha Architecture Handbook*
- *Alpha Architecture Reference Manual*
- *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*

DEC Ada provides a number of predefined floating-point types. Section 1.1.3.1 lists the representation and storage size for each type.

Table 1–2 Representations and Storage Sizes for DEC Ada Predefined Package STANDARD

Ada Type	Representation	Storage Size (Bits)
FLOAT	F_floating[1] IEEE single float[2]	32
LONG_FLOAT	D_floating or G_floating[3] IEEE double float[2]	64
LONG_LONG_FLOAT[4]	H_floating	128
LONG_LONG_FLOAT[5]	G_floating	64

[1]When the value of the pragma FLOAT_REPRESENTATION is VAX_FLOAT (see Section 1.1.3.1); the default on all OpenVMS systems.

[2]When the value of the pragma FLOAT_REPRESENTATION is IEEE_FLOAT (see Section 1.1.3.1); available on Alpha systems only.

[3]The representation of the type LONG_FLOAT depends on the values of the pragmas FLOAT_REPRESENTATION and LONG_FLOAT. You can also use the ACS CREATE LIBRARY, CREATE SUBLIBRARY, and SET PRAGMA commands to control the representation. See Sections 1.1.3.1 and 1.1.3.2 and Chapter 3 of the *DEC Ada Language Reference Manual* for more information about the pragmas. See the *Developing Ada Programs on OpenVMS Systems* for more information about the ACS commands.

[4]On VAX systems.

[5]On Alpha systems.

Table 1–3 Representations and Storage Sizes for DEC Ada Predefined Package SYSTEM

Ada Type	Representation	Storage Size (Bits)
F_FLOAT	F_floating	32
D_FLOAT	D_floating	64
G_FLOAT	G_floating	64
H_FLOAT[1]	H_floating	128
IEEE_SINGLE_FLOAT[2]	IEEE single float	32
IEEE_DOUBLE_FLOAT[2]	IEEE double float	64

[1]On VAX systems only.

[2]On Alpha systems only.

DEC Ada lets you 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 DEC Ada compiler chooses the first of the types STANDARD.FLOAT, STANDARD.LONG_FLOAT, and STANDARD.LONG_LONG_FLOAT (on VAX systems only) that has adequate precision and range and uses it as the parent type from which the new type is derived.

Depending on the default or explicit values of the pragmas FLOAT_REPRESENTATION and LONG_FLOAT, the representations given in Table 1–4 are used if the specified range can also be accommodated. See Section 1.1.3.1 for more information about the pragma FLOAT_REPRESENTATION. See Section 1.1.3.2 for more information about the pragma LONG_FLOAT.

Table 1–4 Representations Chosen for Specified Digits

Value of the Pragma FLOAT_REPRESENTATION	Value of the Pragma LONG_FLOAT	Digits Specified	Representations
VAX_FLOAT	G_FLOAT	1 .. 6	F_floating
		7 .. 15	G_floating
		16 .. 33	H_floating[1]
VAX_FLOAT	D_FLOAT	1 .. 6	F_floating
		7 .. 9	D_floating
		10 .. 33	H_floating[1]
IEEE_FLOAT[2]	Not applicable	1 .. 6	IEEE single float[2]
		7 .. 15	IEEE double float[2]

[1]On VAX systems only.

[2]On Alpha systems only.

For example, in the following declaration the pragma LONG_FLOAT ensures that the D_floating representation of the type LONG_FLOAT is in effect when the declaration is compiled. On VAX systems, the compiler chooses the type STANDARD.LONG_LONG_FLOAT as the parent type for the type SIZE. Although a D_floating representation satisfies the precision, it does not satisfy the range.

```
pragma FLOAT_REPRESENTATION(VAX_FLOAT);
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 model number limits specified by the Ada language determine the choice of representation for a floating-point type. See Chapter 3 of the *DEC Ada Language Reference Manual*. Once the representation is chosen, the full accuracy of the underlying 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 D_floating representation to be used in calculations involving objects of the type:

```
pragma FLOAT_REPRESENTATION(VAX_FLOAT);
pragma LONG_FLOAT(D_FLOAT);
package FLOAT_TYPES is
type VOLUME is digits 9 range -100.0 .. 100.0;
 . . .
end FLOAT_TYPES;
```

Table 1–5 lists the model numbers for each underlying floating-point type (and, thereby, for each DEC Ada predefined floating-point type). The ranges in the table are approximate. The exact ranges are listed in Appendix F of the *DEC 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 1–5 Model Numbers Defined for Each Floating-Point Type

DEC Ada Representations and Types	Digits (D)	Mantissa Bits (B)	Exponent Range (–4*B..+4*B)	Approximate Magnitude
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[1] H_FLOAT LONG_LONG_FLOAT	33	111	–444..444	1.1E–134..4.5E+133
IEEE single float[2] IEEE_SINGLE_FLOAT FLOAT	6	21	–84..84	2.6E–26..1.9E+25
IEEE double float[2] IEEE_DOUBLE_FLOAT LONG_FLOAT	15	51	–204..204	1.9E–62..2.6E+61

[1]On VAX systems only.

[2]On Alpha systems only.

For both predefined and nonpredefined types, the Ada language rules about safe numbers also apply (see Chapter 3 of the *DEC Ada Language Reference Manual*). Table 1–6 lists the safe numbers for each underlying floating-point type (and, thereby, for each DEC Ada floating-point type).

The ranges in the table are approximate. The exact ranges are listed in Appendix F of the *DEC 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.

Table 1–6 Safe Numbers Defined for Each Floating-Point Type

DEC Ada Representations and Types	Digits (D)	Mantissa Bits (B)	Exponent Range (–E..+E)	Approximate Magnitude
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[1] H_FLOAT LONG_LONG_FLOAT	33	111	–16383..16383	8.4E–4933..5.9E+4931
IEEE single float[2] IEEE_SINGLE_FLOAT FLOAT	6	21	–126..127	1.2E–38..4.3E+37
IEEE double float[2] IEEE_DOUBLE_FLOAT LONG_FLOAT	15	51	–1022..1023	2.2E–308..2.3E+307

[1]On VAX OpenVMS systems only.

[2]On Alpha OpenVMS systems only.

1.1.3.1 Pragma FLOAT_REPRESENTATION

The DEC Ada predefined pragma FLOAT_REPRESENTATION acts as a program library switch that controls the representation of the DEC Ada predefined floating-point types STANDARD.FLOAT, STANDARD.LONG_FLOAT, and STANDARD.LONG_LONG_FLOAT. It also controls the representation of types declared with a floating-point definition. You can specify values for this pragma as follows:

- On VAX systems, the only value you can specify is VAX_FLOAT. This value causes floating-point types to be represented by the underlying VAX hardware types: F_floating, D_floating, G_floating, or H_floating.

- On Alpha systems, you can specify either VAX_FLOAT (the default) or IEEE_FLOAT. When you specify VAX_FLOAT, floating-point types are represented by any of the VAX hardware types except H_floating. When you specify IEEE_FLOAT, floating-point types are represented by the IEEE floating-point types: IEEE single float and IEEE double float.

On all OpenVMS systems, when the value of the pragma FLOAT_REPRESENTATION is VAX_FLOAT, the default representation of the type LONG_FLOAT is G_ floating. You can control the representation of the type LONG_FLOAT (or types derived from the type LONG_FLOAT) by using the pragma LONG_ FLOAT (see Section 1.1.3.2).

Use of the pragma FLOAT_REPRESENTATION implies a recompilation of the predefined environment—the package STANDARD—for a given program library. SEE *Developing Ada Programs on OpenVMS Systems* for a discussion of the implications of recompiling the package STANDARD.

For example, the compilation of the following unit causes all subsequent compilations in the same library to use the IEEE floating-point representations to represent floating-point types (the value IEEE_FLOAT valid only on Alpha systems):

```
pragma FLOAT_REPRESENTATION(IEEE_FLOAT);
package USE_IEEE_FLOAT is
   --
   -- Declare some floating-point types
   --
end USE_IEEE_FLOAT;
```

To return to VAX floating-point representations, you can use one of the following methods:

- Compile another unit (in the same library) that contains the pragma FLOAT_REPRESENTATION(VAX_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 OpenVMS Systems* for information on the ACS commands.

1.1.3.2 Pragma LONG_FLOAT

The DEC Ada predefined pragma LONG_FLOAT acts as a program library switch that controls whether to use the G_floating or D_floating representation to represent the type LONG_FLOAT. (By default, the G_floating representation is used.)

Note

The pragma LONG_FLOAT has an effect only when the value of the pragma FLOAT_REPRESENTATION is VAX_FLOAT (the only possible value on VAX systems and the default value on Alpha OpenVMS systems). See Section 1.1.3.1 for more information.

Use of this pragma implies a recompilation of the predefined environment—the package STANDARD—for a given program library. See the *DEC Ada Language Reference Manual* for the specific rules governing the use of this pragma. See *Developing Ada Programs on OpenVMS Systems* for a discussion of the implications of recompiling the package STANDARD.

For example, the compilation of the following unit causes all subsequent compilations in the same library to use the set of representations that include D_floating, as appropriate (see Section 1.1.3):

```
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 (on VAX systems only).
   --
   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:
  1. Deleting it with either the ACS DELETE LIBRARY or DELETE SUBLIBRARY command
  2. Creating it with the ACS CREATE LIBRARY or SUBLIBRARY command

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

1.1.4 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.

DEC Ada supports values of T'SMALL in the range 2.0^{-62} .. 2.0^{31}. In the absence of a length representation clause for T'SMALL, the compiler chooses the largest power of 2 that is less than or equal to the delta value. For example, in the following declaration MY_FIXED'SMALL is 0.03125 (2^{-5}):

```
type MY_FIXED is delta 0.05 range 0.0 .. 1.0;
```

In the presence of a length clause, the compiler uses the value specified in the clause. For example, in the following declaration MY_FIXED'SMALL is 0.05:

```
type MY_FIXED is delta 0.05 range 0.0 .. 1.0;
for MY_FIXED'SMALL use 0.05;
```

The underlying model numbers chosen for the type are evenly spaced multiples of the value of T'SMALL. The set of model numbers is also limited by the range specified in the type declaration.

Values for objects of a fixed-point type are represented in DEC Ada as signed or unsigned, two's complement (binary) numbers that are multipliers of the value of T'SMALL. You can use length representation clauses to achieve unsigned representations. See Section 1.2.3 for more information.

In DEC Ada, the storage size for an object of any fixed-point type is determined by its delta and range, and—if the object is not packed—the storage size is rounded up to an 8-, 16-, or 32-bit boundary. You can change the size with a representation clause (see Section 1.2 of this manual and Chapter 13 of the *DEC Ada Language Reference Manual* for more information). Storage size for the fixed-point type may be affected when you specify the value of T'SMALL in a length representation clause.

Unless the language specifies otherwise, operations on fixed-point types are handled as follows:

- On VAX systems, results that are powers of two are truncated towards 0.0. Results that are not powers of two are rounded.
- On Alpha systems, all results are rounded.

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. When you are working with fixed-point numbers, you may obtain results that in some cases are not what you expect. (See Chapter 4 of the *DEC 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). 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 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 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 it is rounded up to 0.3 when printed out.

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 length representation clause. See Chapter 13 of the *DEC Ada Language Reference Manual* for more information.

1.1.5 Array Types and Objects

In DEC 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 the space may or may not be read or written by operations on the object. 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. The default alignment of DEC Ada array components is as follows:

- On VAX systems, array components are byte-aligned by default.
- On Alpha OpenVMS systems, array components are naturally aligned by default. Natural alignment means that 1-byte components are aligned on byte boundaries, 2-byte components are aligned on 2-byte boundaries, 4-byte components are aligned on 4-byte boundaries, and so on.

You can use the pragma COMPONENT_ALIGNMENT to change the default alignment for any array type (see Section 1.2.2).

To force bit alignment and/or to minimize gaps, use the pragma PACK with the array type declaration (see Section 1.2.1).

Consider the following declarations:

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

Because values of the type COLORS are stored in a byte (see Section 1.1.1), 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);
```

1.1.6 Record Types and Objects

In DEC Ada, the representation chosen for objects of a record type depends on a complex interaction among any applicable representation clauses or pragmas and the types and subtypes of the record components. DEC 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 lets you explicitly specify all of the components of a record object and 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. See Chapter 13 of the *DEC Ada Language Reference Manual* and Sections Section 1.2.6 and Section 1.2.2 of this manual for more information on record component alignments.

In the absence of a record representation clause, record components and subcomponents are aligned by default as follows:

- On VAX systems, byte alignment is the default.
- On Alpha systems, natural alignment is the default. Natural alignment means that 1-byte components are aligned on byte boundaries, 2-byte components are aligned on 2-byte boundaries, 4-byte components are aligned on 4-byte boundaries, and so on.

You can use the pragma COMPONENT_ALIGNMENT to change the default alignment (see Section 1.2.2).

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. Because they are not affected by representation clauses, the variants are laid out starting on the first byte boundary after A.

The type SHOW_LAYOUT from the preceding example can be declared with a representation clause that specifically places one of the variants elsewhere. In this case, that variant is laid out first. In the preceding example, if SHOW_ LAYOUT is declared in the following representation clause, the compiler lays out B first, then DISCRIMINANT, then A, then C:

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

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

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

The dynamic calculation of component offsets and sizes can be done when the type is elaborated, or it can be done later when any of the following occur:

- The subtypes of all of the components are forced
- The type is forced
- The component is selected (This happens when the actual value of a discriminant is needed to make the calculation.)

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 lets the compiler 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 compiler laid out the last component, including any allowances that were made for alignments.

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 an appropriate boundary if necessary). (Rounding depends on whether or not the record type itself is packable; see Section 1.2.1.)

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;
```

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 can 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 can 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, and 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. The estimated storage is rounded up to a byte boundary. Therefore, a total of 88 bytes ((32 + 32 + 625 + rounding bits)/8) is allocated for FLOOR1.

The exact calculation of the size of a record may 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;
```

In addition, the compiler uses simplifyinf 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 differ 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 arrive at one answer. Alternatively, if you ask the compiler for REC_OBJECT'SIZE or REC_OBJECT'MACHINE_SIZE, you receive a different answer. In fact, the compiler bases its answer 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 Chapter 13 of the *DEC Ada Language Reference Manual* for more information on the size attributes.

1.1.7 Access Types and Objects

DEC Ada uses a virtual address to represent the value of an access type. The storage size for this value is 4 bytes (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 4).

Note

Since these addresses do not necessarily point directly to objects of the target, the accessed, or the designated type, it is unsafe (as well as nonportable) to use the predefined generic procedure UNCHECKED_CONVERSION to convert between addresses and access types. Unchecked conversion does not work between machine addresses and access types that point to unconstrained arrays. See Section 9.2 for more information.

Each nonderived access type is associated with a collection, which is storage 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 appropriate boundary) to be allocated for the collection associated with the type.

The collection is not extended if it is exhausted. If you specify a zero or negative value, no storage is allocated for the collection, and the collection is not extended. If you do not specify a 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. DEC Ada does not provide automatic garbage collection. See Section 1.4.2 for more information on storage deallocation. See Chapter 13 of the *DEC Ada Language Reference Manual* for more information on length representation clauses and collections.

In the following example, a 512-byte 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 requires 4 bytes (32 bits) for the integer component NUM.

- Each access type component (NEXT_NUM) requires 4 bytes (32 bits).

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;

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;
```

1.1.8 Address Types and Objects

DEC Ada uses a 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 4 bytes (32 bits).

1.1.9 Task Types and Objects

When you declare an object of a task type, the value of the object is used by the DEC Ada run-time library to determine the address of the task control block created for the task.

The storage size for an object of a task type is as follows:

- On VAX systems, it is 4 bytes (32 bits).
- On Alpha systems, it is 8 bytes (64 bits).

See Chapter 7 for more information on task storage allocation.

1.2 Data Optimization

DEC Ada provides the following to let you tailor the representation of nonpredefined types:

Representation clauses
Address clauses
The language-defined representation pragma PACK
The DEC Ada representation pragma COMPONENT_ALIGNMENT

Type representation clauses and pragmas also let you control the representation of any new or derived types that you declare.

The following sections discuss the individual DEC Ada features available for controlling type representations. When choosing a particular feature, consider the following parameters:

- The level of control that you want or need (where level of control means the ability to specify particular layouts or alignments for specific types).
- Interactions between representation clauses and representation pragmas. Types inherit representation clauses or representation pragmas from parent types, but an explicit pragma can override the effect of an inherited pragma.
- Performance.
- Portability.

Table 1–7 is a general comparison of the DEC Ada features with respect to these parameters. Sections Section 1.2.1 to Section 1.2.7 discuss individual features in more detail.

Table 1–7 Comparison of DEC Ada Features for Controlling Type Representations

Feature	Level of Control	Other Considerations
Pragma PACK see Section 1.2.1	Moderate; minimizes space but may not align types the way you want them aligned. For example, may cause byte alignment on Alpha systems.	Overridden by representation clauses. Overrides the pragma COMPONENT_ALIGNMENT. Changes in alignments may affect performance. Exact effects may or may not be portable.

(continued on next page)

Table 1–7 (Cont.) Comparison of DEC Ada Features for Controlling Type Representations

Feature	Level of Control	Other Considerations
Pragma COMPONENT_ALIGNMENT see Section 1.2.2	Moderate; changes overall alignments of record or array components.	Overridden by representation clauses and the pragma PACK. In programs that are to run on multiple systems, is useful for ensuring that a layout is chosen for optimal performance on each system. Useful for maintaining portability in situations where record or array types need to have a particular layout and alignment (for example, in mixed-language programs or in programs where data is written to or read from files and must have the same layout on all systems).
Representation Clause see Sections 1.2.3 to Section 1.2.6	High; allows you to control specific layout and alignment of data.	Overrides the pragmas PACK and COMPONENT_ ALIGNMENT. May or may not affect performance. May not be portable.

1.2.1 Pragma PACK

The predefined pragma PACK lets you minimize gaps between the components of composite types (record and array types). When you apply the pragma PACK to a DEC Ada record or array type declaration, it has an effect on the record or array components that are packable. It may also have an effect on component alignment.

In DEC 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 may have no effect on an array of record components.

Table 1–8 lists the type categories provided in DEC Ada and shows whether or not components of each type are packable.

Table 1–8 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	Yes
Address	No	Yes
Access	No	Yes
Task	No	Yes
Record	Depends[2]	Only if packable
Array	Depends[3]	Only if packable

[1]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. 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.

[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 1–9 shows the effect of the pragma PACK on arrays and records with packable components.

Table 1–9 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.

(continued on next page)

Table 1–9 (Cont.) Effects of Packing the Components of Arrays and Records

	With Length Representation Clause on Component Type	Without Length Representation Clause on 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 alignment allow (see Sections 1.1.5 and 1.1.6).	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 is not packed, the space-saving benefit of the small range of the SMALL_FIXED_POINT components is lost. In this case, the compiler aligns all components 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 1.2.3 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 the 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.

Because the array is declared without the pragma PACK, all of the components are aligned on byte boundaries, and each component has an effective size of 8 bits instead of 4. The size of the array is 80 bits. For the array PACKED_ SMALL_INTEGER_ARRAY, each component has a size of 4 bits, and any extra space between the components is eliminated. The size of the array is 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.
```

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 1.1.6 and 1.2.5 of this manual and Chapter 13 of the *DEC Ada Language Reference Manual*.

1.2.2 Pragma COMPONENT_ALIGNMENT

DEC Ada provides the pragma COMPONENT_ALIGNMENT, which lets you control the default alignment of array and record components (see Sections 1.1.5 and 1.1.6).

You can use the /WARNINGS=COMPILATION_NOTES qualifier with any of the DEC Ada compilation commands to determine:

- Alignments the compiler has chosen for the array components in your program
- Alignments the compiler has chosen for the record components in your program

You may want to change or ensure the alignment of certain record and array components for various reasons. For example:

- You are working with data that is defined in a specific format (something other than Ada format, Alpha format, and so on) and you need to match it.
- You are concerned about performance. For example, access speed is faster on Alpha systems if your data is naturally aligned. However, if you are working on program with large amounts of data, paging may begin to interfere with performance. In that case, you may want to compress the data and byte align it.
- You are writing mixed-language programs that involve Fortran common blocks.

You can specify a pragma COMPONENT_ALIGNMENT for a specific array or record type or for a declarative part. The alignment choices are as follows:

- COMPONENT_SIZE—produces natural alignment.
- COMPONENT_SIZE_4—produces natural alignment for components with a size of 4 or fewer bytes. Anything larger is aligned on a 4-byte boundary.
- DEFAULT—produces the default alignment for the system you are working on (see Sections 1.1.5 and 1.1.6).
- STORAGE_UNIT—produces byte alignment.

For example, the following declaration uses the pragma COMPONENT_ALIGNMENT to specify natural alignment for the components of the record type FLOAT_REC:

```
type FLOAT_REC is record
   SINGLE: FLOAT;
   DOUBLE: LONG_FLOAT;
end record;
pragma COMPONENT_ALIGNMENT(COMPONENT_SIZE, FLOAT_REC);
```

In Example 1–1, the pragma COMPONENT_ALIGNMENT applies to all of the record or array declarations in the declarative part (unless they are already specified in another representation pragma or representation clause).

Example 1–1 Using the Pragma COMPONENT_ALIGNMENT

```
package ALIGNMENT_EXAMPLE is

   -- No pragma applies.
   --
   type ARR_NO_PRAGMA is array(1 .. 10) of INTEGER;

   -- Pragma 1; specifies byte alignment.
   --
   pragma COMPONENT_ALIGNMENT(STORAGE_UNIT);

   -- Pragma 2 applies.
   --
   type REC_COMP_SIZE is
      record
         C1 : CHARACTER;          -- at byte 0
         C2 : SHORT_INTEGER;      -- at byte 2
         C3 : LONG_FLOAT;         -- at byte 8
      end record;
   --
   -- Pragma 2; specifies natural alignment.
   --
   pragma COMPONENT_ALIGNMENT(COMPONENT_SIZE, REC_COMP_SIZE);

   -- Pragma 1 applies.
   type REC_STOR_UNIT is
      record
         C1 : CHARACTER;          -- at byte 0
         C2 : SHORT_INTEGER;      -- at byte 1
         C3 : LONG_FLOAT;         -- at byte 3
      end record;

   -- Pragma 3 (in private part) applies.
   --
   type REC_COMP_SIZE_4 is
      record
         C1 : CHARACTER;          -- at byte 0
         C2 : SHORT_INTEGER;      -- at byte 2
         C3 : LONG_FLOAT;         -- at byte 4
      end record;

   -- Pragma 1 applies.
   --
   type ARR_STOR_UNIT is array(1 .. 10) of INTEGER;

   -- Pragma 1 applies.
   --
   type REC_STOR_UNIT_ALSO is
      record
         C1 : CHARACTER;          -- at byte 0
         C2 : ARR_STOR_UNIT;      -- at byte 1
      end record;
```

(continued on next page)

Example 1–1 (Cont.) Using the Pragma COMPONENT_ALIGNMENT

```
private

   -- Pragma 3.
   --
   -- Specifies that components with a size of 4 or fewer bytes
   -- are naturally aligned; components that are larger than 4 bytes
   -- are placed on the next 4-byte boundary.
   --
   pragma COMPONENT_ALIGNMENT(COMPONENT_SIZE_4, REC_COMP_SIZE_4);

end ALIGNMENT_EXAMPLE;
```

When the pragma PACK and the pragma COMPONENT_ALIGNMENT are directly applied to the same type, the pragma PACK takes precedence over the pragma COMPONENT_ALIGNMENT.

Derived types inherit any representation pragmas or clauses that apply to their parent types, but an explicit pragma applied to the derived type takes precedence over an inherited pragma. An explicit pragma COMPONENT_ ALIGNMENT takes precedence over an inherited pragma PACK and vice versa. Example 1–2 shows how the pragmas PACK and COMPONENT_ ALIGNMENT interact.

For more information about the pragma PACK, see Section 1.2.1 and Chapter 13 of the *DEC Ada Language Reference Manual*. For more information about the pragma COMPONENT_ALIGNMENT, see Chapter 13 of the *DEC Ada Language Reference Manual*. For more examples of situations where the pragma COMPONENT_ALIGNMENT is used, see Sections Section 4.6 and Section 4.7.

1.2.3 Length Representation Clauses

Length representation clauses let you 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*2.0**(-8);
for UNSIGNED_FIXED_POINT'SIZE use 8;
```

Example 1–2 Interaction Between the Pragmas PACK and COMPONENT_ALIGNMENT

```
package INTERACTION_EXAMPLE is

   -- Pragma COMPONENT_ALIGNMENT 1.
   --
   pragma COMPONENT_ALIGNMENT(COMPONENT_SIZE);

   -- Pragma COMPONENT_ALIGNMENT 1 applies, causing COMPONENT_SIZE
   -- (natural) alignment.
   --
   type REC_COMP_SIZE is
      record
         C1 : CHARACTER;        -- at byte 0
         C2 : SHORT_INTEGER;    -- at byte 2
         C3 : LONG_FLOAT;       -- at byte 8
      end record;

   -- Pragma PACK applies.
   --
   type REC_PACKED is
      record
         C1 : CHARACTER;        -- at byte 0
         C2 : SHORT_INTEGER;    -- at byte 1
         C3 : LONG_FLOAT;       -- at byte 3
      end record;
   pragma PACK(REC_PACKED);

   -- Pragma PACK applies.
   --
   type REC_PACKED_2 is new REC_PACKED;

   -- Pragma COMPONENT_ALIGNMENT 2 applies, causing COMPONENT_SIZE_4
   -- alignment.
   --
   type REC_COMP_SIZE_4 is new REC_PACKED;
      -- C1 is at byte 0
      -- C2 is at byte 2
      -- C3 is at byte 4
   --
   -- Pragma COMPONENT_ALIGNMENT 2.
   --
   pragma COMPONENT_ALIGNMENT(COMPONENT_SIZE_4, REC_COMP_SIZE_4);

end INTERACTION_EXAMPLE;
```

The first declaration causes objects of the type UNSIGNED_INTEGER to be stored as unsigned 2-byte quantities, rather than as signed 4-byte quantities. The second declaration causes objects of the type UNSIGNED_FIXED_POINT to be stored as unsigned bytes, rather than as signed 2-byte quantities. Because of Ada language rules, arithmetic operations involving these objects are signed.

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 overrides the pragma PACK and causes the size of each component in the packed array to be 4 bits.

Chapter 13 of the *DEC Ada Language Reference Manual* gives complete information on the use of length representation clauses. See Table 1–9 for information on the interaction between length representation clauses and the pragma PACK.

1.2.4 Enumeration Representation Clauses

Enumeration representation clauses let you 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;
```

The storage allocated for MY_UNSIGNED_ANSWER is 2 bytes (16 bits). Even though only three integer codes must be represented, 2 bytes (16 bits) are 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. If you redeclare the type ANSWER as follows, the internal codes will be signed:

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

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 4 bytes (32 bits).

1.2.5 Record Representation Clauses

Record representation clauses let you force a record type to have a particular representation. They are useful any time 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.
- You can use record representation clauses to lay out record types that declare objects that may be passed to other-language routines, operating-system routines, or 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, which contains four 4-bit fields:

```
-- Record defining a 4-bit field.
--
type FIELD_TYPE is
   record
      BIT1 : BOOLEAN;
      BIT2 : BOOLEAN;
      BIT3 : BOOLEAN;
      BIT4 : BOOLEAN;
   end record;

for FIELD_TYPE'SIZE use 4;
for FIELD_TYPE use
   record
      BIT1 at 0 range 0 .. 0;
      BIT2 at 0 range 1 .. 1;
      BIT3 at 0 range 2 .. 2;
      BIT4 at 0 range 3 .. 3;
   end record;

-- Record defining a 16-bit mask; the record
-- representation clause lays out the 4-bit fields
-- so that they are contiguous half-bytes.
--
type BIT_MASK_TYPE is
   record
      FIELD1 : FIELD_TYPE;
      FIELD2 : FIELD_TYPE;
      FIELD3 : FIELD_TYPE;
      FIELD4 : FIELD_TYPE;
   end record;
```

```
for BIT_MASK_TYPE use
   record
      FIELD1 at 0 range 0 .. 3;
      FIELD2 at 0 range 4 .. 7;
      FIELD3 at 1 range 0 .. 3;
      FIELD4 at 1 range 4 .. 7;
   end record;

for BIT_MASK_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;
```

The representation clause on the type VARIANT_RECORD causes the discriminant, 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 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;
```

The compiler can set up the layout for the type SLOW_RECORD 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.C, the size of SLOW_OBJECT.B must be calculated, 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;
```

FASTER_OBJECT is 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.

When you use a record representation clause to request a small storage space for a component of a nonfixed-point discrete type, the record component value may be biased. (Its value can be altered predictably.)

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.

In the following example, the values of R.C are biased so that they can be stored in the 2 bits required by the record representation clause. Without the record representation clause, they are each stored in 32 bits:

```
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;
 . . .
```

1.2.6 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 *DEC Ada Language Reference Manual* gives the syntax and rules for using alignment clauses.

In DEC Ada, records can be aligned on any byte address that is a power of 2, up to 512 (or 2^9). In the following fragment, the value of ALIGN_AT can be any integer in the series 1, 2, 4, 8, . . . , 512. A value of 1 indicates byte alignment, a value of 2 indicates 2-byte alignment, and a value of 512 indicates 512-byte 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 is 16, then the components of the record SHOW_ALIGNMENT are aligned on 16-byte boundaries.

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

Alignment clauses can be useful in a mixed-language environment where you may want to force objects to particular boundaries. However, the VAX hardware generally requires little alignment:

- Only a few instructions and OpenVMS Run-Time Library routines need alignments (for example, queue and interlocked instructions).
- DEC Ada currently generates few interlocked instructions.

The Alpha hardware runs more efficiently with naturally aligned data.

The DEC Ada pragma COMPONENT_ALIGNMENT is also useful in a mixed-language environment. See Section 1.2.2 for more information.

1.2.7 Address Clauses

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

- A program should not use address clauses to overlay two or more Ada objects.
- When you declare an object with an address clause, the usual implicit or explicit initialization associated with the type of the object is performed. See Section 3.2.1 of the *DEC Ada Language Reference Manual* for the rules about intializing objects. Access values are initialized to **null**, and record components can also receive initial values.

For example, consider the following declarations:

```
type HEADER_TYPE is record
   HEADER_FIELD: BYTE_ARRAY(1 .. LENGTH_OF_FIELD);
   FILLER      : BYTE_ARRAY(1 .. LENGTH_OF_FILLER)
                    := (others => SLASH);
end record;

X: HEADER_TYPE;
for X use at SOME_ADDRESS;
```

Because the component FILLER is initialized by declaring the type HEADER_ TYPE, you might expect X.FILLER to be initialized to the same value. Instead, because the DEC Ada compiler is following the Ada language rule about initializing X when it is declared with the address clause, both X.HEADER_ FIELD and X.FILLER are initialized with a null value chosen by the compiler. The type initialization is overwritten by the object initialization.

To control the initialization of X and still have the effect of an address clause, you can rewrite the previous code as follows:

```
type AHT is access HEADER_TYPE;
for AHT'STORAGE_SIZE use 0;

function TO_AHT is new UNCHECKED_CONVERSION(ADDRESS, AHT);

X: HEADER_TYPE renames TO_AHT(SOME_ADDRESS).all;
```

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. When you use address clauses, you need to be sure that you choose values that are meaningful in the OpenVMS environment.

One way to obtain a meaningful value is to use the OpenVMS Run-Time Library (RTL) routine LIB$GET_VM to obtain a storage location. Example 1–3 is a complete procedure showing the use of an address clause to overlay an Ada record object onto storage allocated by LIB$GET_VM. The *OpenVMS RTL Library (LIB$) Manual* describes LIB$GET_VM in more detail.

Example 1–3 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 OpenVMS 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;
```

(continued on next page)

Example 1–3 (Cont.) Using an Address Clause and LIB$GET_VM

```
   -- 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..";

   -- 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;
```

1.3 Determining the Sizes of Types and Objects

DEC Ada provides a number of methods for determining how much storage is 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 other structures are 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, and the other two are described in *Developing Ada Programs on OpenVMS 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 *DEC 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 different. For example, given the following declarations, the value of BOOL′SIZE is 1, and the value of B′SIZE is 8:

```
type BOOL is new BOOLEAN;
B: BOOL;
```

One bit is the minimum amount of storage required for an object of the type BOOL. Eight bits is the actual amount of storage used by the object B.

The DEC Ada attribute T′MACHINE_SIZE provides similar information for a type or subtype that O′SIZE provides for an object. Table 1–10 summarizes the differences between T′SIZE, O′SIZE, and T′MACHINE_SIZE.

Table 1–10 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-, 32-, or 64-bit boundary (64-bit boundaries apply to discrete types on AXP systems only). Representation is signed.

(continued on next page)

Table 1–10 (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	Actual 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-, 32-, or 64-bit boundary (64-bit boundaries apply to discrete types on AXP systems only). 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 or subtype T. Result is the actual number of bits used, rounded up to a byte boundary.

The T'MACHINE_SIZE of a base type can be equal to or greater than the T'SIZE of the same base type. 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. The values of INT8'MACHINE_SIZE and I'SIZE are equal.

Table 1–11 gives a set of results for a variety of interesting cases.

Table 1–11 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	17
Object B'SIZE	17
Type BOOL17'MACHINE_SIZE	32
Type BOOL17'BASE'SIZE	17
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

(continued on next page)

Table 1–11 (Cont.) Results of Size Attributes for Various Types and Objects

type NET **is new** ET **range** 0 .. 7; NE: NET;	
Type NET'SIZE	8
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;	
Type NT'SIZE	8
Object N'SIZE	8
Type NT'MACHINE_SIZE	8
Type NT'BASE'SIZE	32
Type NT'BASE'MACHINE_SIZE	32
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

1.4 Storage Allocation and Deallocation

To make efficient use of storage from your DEC 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.

1.4.1 Storage Allocation

The DEC Ada compiler stores objects in registers, on a stack, in static memory, or in dynamic memory (on the heap) depending upon the objects' sizes, when their sizes are known, their types, 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 it from the time the object is elaborated until the time when its containing scope is left. See Chapter 9 for more information on working with address values. See Chapter 9 of the *DEC Ada Language Reference Manual* and Chapter 7 of this manual for more information on the pragma VOLATILE.

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 1.4.2 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 Chapter 13 of the *DEC Ada Language Reference Manual* for more information. See Section 1.1.7 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. (T'STORAGE_SIZE specifies the number of bytes to be used for a collection. In DEC Ada, this number is rounded up to an appropriate boundary). For example, the values 512*4, 512*8, and 512*12 are better than the values

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 results in reduced fragmentation of memory. Also, when you free several collections (implicitly) at scope exit, the freed storage is likely in blocks large enough to be useful for other collections.

1.4.2 Storage Deallocation

DEC 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 *DEC 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. You conserve the use of the collection, rather than deallocating the collection for general use by your program.

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 1–4 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 1–4 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 8*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 1–4 (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 1–4 (Cont.) Using UNCHECKED_DEALLOCATION to Control Access Type Storage Deallocation

```
   -- Concurrently, create (and deallocate) the series of
   -- 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;
```

2
Input-Output Facilities

Although DEC Ada lets you invoke OpenVMS input-output system services and Record Management Services (RMS) directly (see Chapters Chapter 4 and Chapter 5), for most applications it is not necessary to do so. The DEC 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.

DEC 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

The packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO are predefined by the Ada language. The other packages are predefined by the DEC 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 *DEC Ada Language Reference Manual*.

The DEC Ada predefined packages and their operations are implemented using 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 about input–output in the *DEC Ada Language Reference Manual*. You should also be familiar with the following:

- RMS file organizations and access methods
- How to work with OpenVMS file specifications and directories

- The OpenVMS File Definition Language (FDL)

If you need introductory information on OpenVMS file specifications and directories or FDL, see the *Guide to OpenVMS File Applications*. For more information about RMS and RMS services, see the *OpenVMS Record Management Services Reference Manual*. For more information on FDL, see the *OpenVMS Record Management Utilities Reference Manual*.

2.1 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 DEC Ada input-output packages supply CREATE and OPEN procedures that let you 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.

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.text and associate it with
   -- the file object ONE_FILE.
   --
   CREATE (FILE => ONE_FILE,
           NAME => "external_file.text");
    . . .
   -- Close external_file.text and disassociate it with
   -- the file object ONE_FILE.
   --
   CLOSE  (ONE_FILE);
    . . .
   -- Reopen external_file.text and associate it with
   -- a different file object.
   --
   OPEN   (FILE => ANOTHER_FILE,
           MODE => OUT_FILE,
           NAME => "external_file.text");
    . . .
end MAKE_FILES;
```

When you create or open a DEC Ada file object, the external file with which it is associated is an 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 an 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 2.3 discusses the FORM parameter and system-dependent external file attributes in more detail. Sections Section 2.6.1 to Section 2.7 provide tables of default attributes for each DEC Ada input-output package.

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

2.1.1 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.

The packages SEQUENTIAL_IO and SEQUENTIAL_MIXED_IO provide sequential access to Ada sequential files. See Section 2.6.1 for more information about sequential files.

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

2.1.2 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 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, open, or reset the file. The current index determines which element is involved in the next read or write operation.

The packages DIRECT_IO and DIRECT_MIXED_IO provide direct access to Ada direct files. See Section 2.6.2 for more information about direct files.

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

2.1.3 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 is 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.

The packages RELATIVE_IO and RELATIVE_MIXED_IO provide relative access to Ada relative files. See Section 2.6.3 for more information about relative files.

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

2.1.4 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.

The packages INDEXED_IO and INDEXED_MIXED_IO provide indexed access to Ada indexed files. See Section 2.6.4 for more information about indexed files.

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

2.1.5 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 INTEGER'LAST. (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 1 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.

The package TEXT_IO provides sequential access to Ada text files. See Section 2.7 for more information about text files.

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

2.2 Naming External Files

In DEC Ada, you identify external files using OpenVMS file specifications. All of the DEC Ada input-output packages have CREATE and OPEN procedures, which have a NAME parameter that lets you associate the name of an external file with a particular file object. The NAME parameter can have one of the following values:

- A string that denotes an OpenVMS 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. 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 lets you identify an external file (see Section 2.3). In DEC Ada, the FORM parameter takes as its value an OpenVMS 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 is used by default if any of that information is omitted from the string given for the NAME parameter. In the following example, the external file has 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 *OpenVMS User's Manual* and the *Guide to OpenVMS File Applications*.

2.2.1 File Specification Syntax

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

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

Note

You can access files that reside on non-OpenVMS systems by enclosing the name of the file (in its required format) in a quoted string. See the *OpenVMS User's Manual* for more information.

node
The name of a network node. This element applies only to systems that are part of a network (systems that support DECnet).

device
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
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
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
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
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 *OpenVMS User's Manual*.

You can use DEC 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 2.3.3.

2.2.2 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. Logical names are a convenient shorthand for file names to which you refer frequently. See the *OpenVMS User's Manual* and *Guide to OpenVMS 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 *OpenVMS DCL Dictionary*.

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 *OpenVMS User's Manual*.

By default, OpenVMS creates a set of logical names for you when you log in. Table 2–1 lists the predefined names that are most relevant to DEC Ada input-output.

Table 2–1 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 2–1 (Cont.) Predefined (Default) Logical Names

Logical Name	Table in Which the Name is Stored	What the Name Represents
SYS$NET	Process	The source process that invokes a target process in DECnet task-to-task communication. When opened by the target process, SYS$NET represents the logical link over which that process can exchange data with its partner. SYS$NET is defined only during task-to-task communication. (Task-to-task communication refers to tasks that are OpenVMS images running in the context of a process, not Ada tasks.)
SYS$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 *OpenVMS User's Manual* explains and demonstrates the use of these names. Table 2–2 shows the source of the equivalence strings associated with them.

Table 2–2 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* is the device name of your terminal; *disk* is the batch input or command file; and *log file* is the batch job log file.

2.3 Specifying External File Attributes

The CREATE and OPEN procedures in the DEC Ada input-output packages all have a FORM parameter that lets you specify the system-dependent attributes of an external file. Most of the time you do 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 2.3.3). 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 an OpenVMS FDL string or 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 RMS attributes available in FDL, and you thereby supersede the default file attributes of your input-output package (see Section 2.3.3).

If you are not familiar with FDL, see the *Guide to OpenVMS File Applications*. It introduces FDL and shows how to design files using the Edit/FDL Utility. See the *OpenVMS Record Management Utilities Reference 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 DEC Ada FORM parameters.

2.3.1 The OpenVMS File Definition Language (FDL): Primary and Secondary Attributes

FDL statements—whether in an FDL file or in a DEC Ada form string—specify predefined 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 2–3 lists the available primary and secondary attributes.

Table 2–3 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 2–3 (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 happens when it is opened or closed; whether or not data checking is 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
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 are not useful in an Ada FORM parameter.	BACKUP, CREATION, EXPIRATION, REVISION

(continued on next page)

Table 2–3 (Cont.) FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
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 2–3 (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 2–3 (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; appears 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 2–3 (Cont.) FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
ANALYSIS_OF_KEY	Result of using Analyze/RMS_File Utility; appears 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 raises the Ada predefined exception USE_ERROR.

- The primary attributes must appear in the order shown in Table 2–3.
- 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. So, 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. 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 (" "). 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, DEC Ada also lets you give a reference to an FDL file using a OpenVMS 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 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 OpenVMS File Applications* for more information on the Edit/FDL Utility and file design.

Table 2–4 describes the primary and secondary FDL attributes that you are most likely to use in a DEC 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 *OpenVMS Record Management Utilities Reference Manual*. See that manual for details.

As shown in Table 2–4, 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 32-bit decimal integer (based integers or underscores are not allowed).

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

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. 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 2–4 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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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 are initially allocated for the file. If 0, the system does 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.
BUCKET_SIZE	Integer in the range 0 to 63. Default value is 0.	Sets the number of blocks per bucket. If 0, 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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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 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 RMS statistics gathering for subsequent use in doing performance analysis.
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.)

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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).
SEQUENTIAL_ONLY	Switch. Default value is NO.	Indicates that the file can only be processed sequentially, 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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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.
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 2.7.4 of this manual and the *OpenVMS Record Management Utilities Reference 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 *OpenVMS Record Management Utilities Reference 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 does not impose a maximum record length. The records in an indexed file, however, cannot cross bucket boundaries.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
		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. 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 *OpenVMS Record Management Utilities Reference Manual* for the maximum sizes allowed for the various record organizations and formats.
ACCESS		
DELETE	Switch. The default value is FALSE.	Permits 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 RMS get or find operations.
PUT	Switch. PUT when creating a file.	Permits RMS put or extend operations.
TRUNCATE	Switch. Default value is FALSE.	Permits RMS truncate operations.
UPDATE	Switch. Default value is FALSE.	Permits RMS update or extend operations.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
SHARING		
DELETE	Switch. No default value.	Lets other users delete records from the file.
GET	Switch. TRUE if ACCESS GET has also been specified.	Lets other users 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 DEC 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.	Lets other users write records to the file.
UPDATE	Switch. No default value.	Lets other users update records that currently exist in the file.
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, RMS chooses a default value (see the *OpenVMS Record Management Utilities Reference Manual*). This attribute is ignored for DECnet operations.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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 *OpenVMS Record Management Utilities Reference Manual* for more information.
TIMEOUT_ENABLE	Switch. No default value.	Specifies the maximum time, in seconds, that are 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, RMS returns an error status. This attribute is ignored for DECnet operations.
TIMEOUT_PERIOD	Integer in the range 0 to 255. No default value.	Specifies the maximum number of seconds that an 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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
TRUNCATE_ON_PUT	Switch. No default value.	Specifies that an 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 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 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 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.
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 *OpenVMS Record Management Utilities Reference Manual* for more information.
AREA	This attribute and its secondary attributes apply only to files with indexed organization. See the *OpenVMS Record Management Utilities Reference Manual* for details.	

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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 an RMS update operation. A value of YES for primary keys is an error; a value of YES for alternate keys is allowed.
DATA_KEY_ COMPRESSION	Switch. Default value is YES.	Controls whether leading and trailing repeating characters in the primary key 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.
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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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 results in an error.
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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
NULL_VALUE	Character or unsigned decimal integer representing an ASCII value. Default value is the ASCII null character (0).	Defines the null value that instructs 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.
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 *OpenVMS Record Management Utilities Reference 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.

(continued on next page)

Table 2–4 (Cont.) Commonly Used FDL Attributes

FDL Attributes	Kind of Value and Default	Function
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.
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 *OpenVMS Record Management Utilities Reference Manual* for more information.

Certain FDL attributes can significantly improve application performance. If the files used by the application are designed and tuned properly, the application runs more efficiently, often because a minimum number of input-output operations 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 2–4 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 2–4 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 OpenVMS File Applications* for more information.

2.3.2 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 OpenVMS 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 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. DEC 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.

2.3.3 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. 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 2.3).

Sections Section 2.6.1 to Section 2.7 provide tables of default attributes for each DEC Ada input-output package. Default external files have the following 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 2.3.2).
- Many FDL default attributes are applied automatically, but they are not shown in the default attribute tables. See the *OpenVMS Record Management Utilities Reference Manual* for the FDL defaults. The DEC 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.

2.4 File Sharing

File sharing in DEC Ada enables concurrent access to the same external file. File sharing permits multiple file objects to be associated with the same external file. File sharing can take place in the same OpenVMS process or across multiple processes. You can share external files for reading, writing, or modifying.

Because DEC Ada files are layered on RMS file organizations, the rules that apply to read and write sharing of RMS files also apply to Ada files. The *Guide to OpenVMS File Applications* gives complete information on file sharing in the OpenVMS environment. For descriptions of the organizations chosen for Ada files, see Section 2.1.

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, DEC Ada uses the MODE parameter to select appropriate default ACCESS and SHARING secondary attributes (see Section 2.3.3 and Tables Table 2–5 through Table 2–13). 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 2–9.
2. If the table does not show a default for a particular attribute, check Table 2–4 or the *OpenVMS Record Management Utilities Reference 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:

- 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 raises 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 OpenVMS File Applications* for more information.

In Example 2–1, read sharing is desired for the relative file REL_FILE.

Example 2–1 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,        1
          NAME => "REL_FILE.DAT",
          FORM => "RECORD;"     &
                     "SIZE 30;" &
                  "SHARING;"    &    2
                     "GET YES;");
 . . .
end CREATE_RELATIVE;

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

(continued on next page)

Example 2–1 (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;                    3
    . . .
   OPEN(FILE => IO_FILE,
        MODE => IN_FILE,
        NAME => "REL_FILE.DAT",        4
        FORM => "RECORD;"       &
                  "SIZE 30;"    &
                "SHARING;"      &      5
                  "PUT YES;");
    . . .
   CLOSE(IO_FILE);
end SHARE_RELATIVE;
```

Key to Example 2–1:

1 The CREATE statement creates a relative, in-out file. DEC Ada gives it the following attributes by default (see Table 2–9):

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 2–4).

2 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.

3 The procedure SHARE_RELATIVE calls the procedure CREATE_ RELATIVE. Because CREATE_RELATIVE does not close REL_FILE.DAT, the file is still open and needs to be shared when SHARE_RELATIVE tries to access it.

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

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

2.5 Record Locking

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

In DEC 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, 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 fails, 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 the *DEC 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;
 . . .
```

2.6 Binary Input-Output

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

- The first kind—SEQUENTIAL_IO, DIRECT_IO, RELATIVE_IO, and INDEXED_IO—lets you 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—lets you 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 2–2 and Figure 2–1 show the use of a mixed-type file (using the package DIRECT_MIXED_IO). The circled numbers in Figure 2–1 match statements in the program EXPENSE_ACCOUNT (Example 2–2) to elements in the file EXPENSES. Figure 2–2 shows the use of a file with elements of the same type (using the package DIRECT_IO).

Sections Section 2.1.1 to Section 2.1.5 describe the structure of DEC Ada files and give their relationship to RMS files. Chapter 14 of the *DEC 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 OpenVMS File Applications*.

Example 2–2 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, "01-08-91");   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 2–1 Using a Mixed-Type File

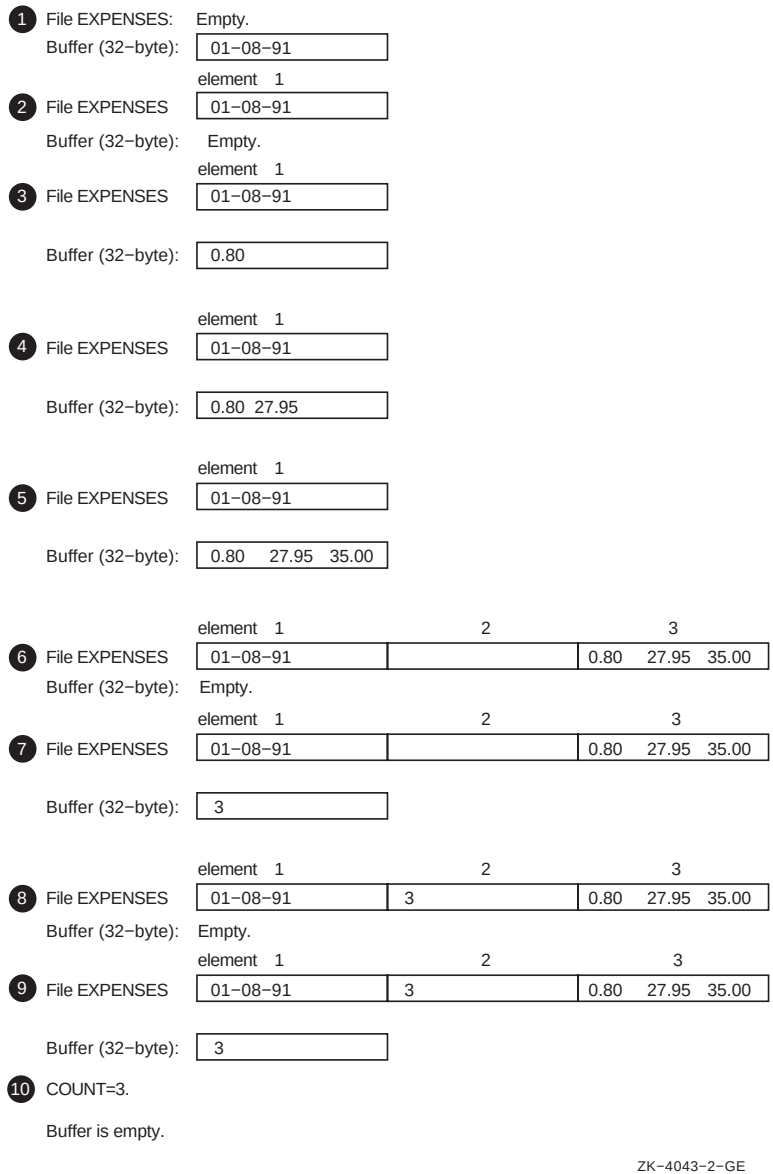
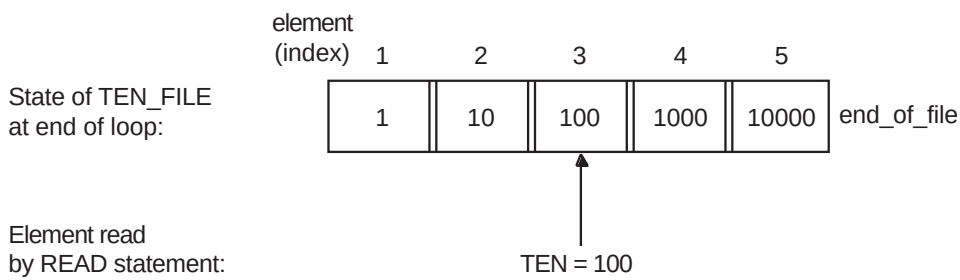


Figure 2–2 Using a Uniform-Type File

```
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.data");
   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

2.6.1 Sequential File Input-Output

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

When you create a file with the package SEQUENTIAL_IO, DEC Ada gives it the default attributes listed in Table 2–5. When you create a file with the package SEQUENTIAL_MIXED_IO, DEC Ada gives it the default attributes listed in Table 2–6. You can use the operations in the packages SEQUENTIAL_IO and SEQUENTIAL_MIXED_IO to open and read files of any RMS organization.

Table 2–5 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 (However, there are physical limitations to SIZE; see the *OpenVMS Record Management Services Reference 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 2–6 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; however, SIZE has physical limitations; see the *OpenVMS Record Management Services Reference 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 2–3 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 2–1 and Examples Example 2–4 and Example 2–7 for examples of using the item input-output operations.

Example 2–3 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;
```

2.6.2 Direct File Input-Output

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

When you create a file with the package DIRECT_IO, DEC Ada gives it the default file attributes listed in Table 2–7. When you create a file with the package DIRECT_MIXED_IO, DEC Ada gives it the default file attributes listed in Table 2–8. 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 is 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 2–7 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 2–8 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 2–4 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 2–2.

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 is unpredictable.

Example 2–4 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;
```

(continued on next page)

Example 2–4 (Cont.) Using the Package DIRECT_MIXED_IO

```
   -- 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.

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

(continued on next page)

Example 2–4 (Cont.) Using the Package DIRECT_MIXED_IO

```
   -- 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;
```

2.6.3 Relative File Input-Output

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

When you create a file with the package RELATIVE_IO, DEC Ada gives it the default file attributes listed in Table 2–9. When you create a file with the package RELATIVE_MIXED_IO, DEC Ada gives it the default file attributes listed in Table 2–10. 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 is 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 2–9 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 2–10 RELATIVE_MIXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	RELATIVE
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	VARIABLE
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 2–5 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 is 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 2–1 and Examples Example 2–4 and Example 2–7 for examples of using the item input-output operations.

Example 2–5 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 2–5 (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;
```

2.6.4 Indexed File Input-Output

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

When you create a file with the package INDEXED_IO, DEC Ada gives it the default file attributes listed in Table 2–11. When you create a file with the package INDEXED_MIXED_IO, DEC Ada gives it the default file attributes listed in Table 2–12. 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 is 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). There is no default bucket size. If you do not specify a bucket size with the FORM parameter, RMS calculates the bucket size based on the maximum record size. (The default is 0).

Table 2–11 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; however, SIZE is also limited by the bucket size; see the *OpenVMS Record Management Services Reference 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 2–12 INDEXED_MIXED_IO: Default File Attributes

File Attribute	Default Value
FILE	
ORGANIZATION	INDEXED
RECORD	
CARRIAGE_CONTROL	CARRIAGE_RETURN
FORMAT	VARIABLE
SIZE	0 (the record size is unlimited; however, the record size is limited by the bucket size; see the *OpenVMS Record Management Services Reference 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 2–6 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 DEC 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 *DEC 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. The FDL CONNECT EQUAL_NEXT and CONNECT_NEXT attributes never have an effect when you are using a READ_BY_KEY procedure.

Example 2–6 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 2–6 (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 2–6 (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 2–6 (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);
```

(continued on next page)

Example 2–6 (Cont.) Using the Package INDEXED_IO

```
    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 2–7 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 2–7 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);
```

(continued on next page)

Example 2–7 (Cont.) Using the Package INDEXED_MIXED_IO

```
   -- 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);

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);
```

(continued on next page)

Example 2–7 (Cont.) Using the Package INDEXED_MIXED_IO

```
-- 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);

-- 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);
```

(continued on next page)

Example 2–7 (Cont.) Using the Package INDEXED_MIXED_IO

```
-- 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;
```

2.7 Text Input-Output

DEC 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, DEC Ada gives it the defaults listed in Table 2–13.

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 IS raised.

Table 2–13 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; however, the record size has physical limitations; see the *OpenVMS Record Management Services Reference 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
WRITE_BEHIND	YES if MODE is OUT_FILE

As shown in Table 2–13, DEC Ada text files are implemented as RMS sequential files. Each line in a text file corresponds to a single RMS record. DEC Ada text files are not stream files.

Although DEC Ada creates text files with variable-length records by default, you can use the FORM parameter (see Section 2.3) 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).

You can also use the FORM parameter to create text files with lines of indefinite length, including lengths greater than the maximum RMS record size. DEC Ada recognizes files with the following characteristics as files of indefinite line length:

- The print form of carriage control
- A 2-byte header size (applies to all records in the external file)
- Variable-length with fixed-length control field (VFC) record format
- A maximum record size of zero

To create a DEC Ada text file with lines of indefinite length, use a FORM parameter in the TEXT_IO.CREATE procedure and specify these characteristics (either explicitly or by relying on defaults). For example:

```
CREATE (FILE => INDEFINITE_LINE_LENGTH_FILE,
        FORM => "RECORD;"                          &
                   "CARRIAGE_CONTROL       PRINT;"  &
                   "CONTROL_FIELD          2;"      &
                   "FORMAT                 VFC;"    &
                   "SIZE                   0;");
```

If you specify a nonzero record size, your text file has lines of the length specified. (The record size must be within the normal range of values for the length of an RMS record.)

Because the "CARRIAGE_CONTROL PRINT" statement gives a default control field size of 2 bytes and a VFC format, you could also use the following form string to create a text file with lines of indefinite length:

```
CREATE (FILE => INDEFINITE_LINE_LENGTH_FILE,
        FORM => "RECORD;"                          &
                   "CARRIAGE_CONTROL       PRINT;"  &
                   "SIZE                   0;");
```

Lines are written to files with indefinite line length as one or more RMS records. The characters in each record's 2-byte header keep track of which records comprise the beginning, middle, and end of a line.

In some cases you may wish to open a text file that has the characteristics of an indefinite-line-length file (for example a file created by some other OpenVMS-related software). If you do not want the file to be treated as

one with indefinite line length, then open the file with the TEXT_IO.OPEN procedure and specify a nonzero record length in the form string. For example:

```
TEXT_IO.OPEN(
   FILE => FIXED_LINE_LENGTH_FILE,
   MODE => IN_FILE,
   NAME => "FIXED_FILE.DAT",
   FORM => "RECORD;"
             "SIZE                  1;");
```

Regardless of its value, the only effect of the nonzero record length in this case is to prevent the file from being treated as one with indefinite line length.

2.7.1 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:

- DEC Ada TEXT_IO operations are implemented with RMS input-output operations, and RMS operations always involve complete records.
- Buffering is used in both terminal input and output (see Section 2.7.3).
- Terminal input is not processed until a line (an 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 2.7.2.) The difference in interpretation can cause a difference in effect.

Example 2–8 shows the use of TEXT_IO operations to write text from a terminal to a file. Sections Section 2.7.1.1 to Section 2.7.1.4 discuss a number of coding methods for accomplishing interactive terminal input-output.

Example 2–8 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
```

(continued on next page)

Example 2–8 (Cont.) Using the Package TEXT_IO

```
   CREATE(MY_COPY, OUT_FILE, "MYCOPY.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));


      -- 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 MYCOPY.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 DEC Ada TEXT_IO operation behaves exactly as it is described in the *DEC 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 OpenVMS 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 2.7.2), 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 returns the value FALSE. However, another call to GET_LINE reads 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 *DEC Ada Language Reference Manual*.

When you do a SKIP_LINE operation in DEC Ada (or any operation that, in effect, does a SKIP_LINE, such as a GET_LINE. See Chapter 14 of the *DEC 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.

2.7.1.1 Line-Oriented Method

Example 2–9 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 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 let the user 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 2–9 declares the variable TERMINAL for this purpose.

Example 2–9 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 2–9, 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. An 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.

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 2–9 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).
   --
   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 2–9 (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;
```

2.7.1.2 Data-Oriented Method

Example 2–10 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 2–10 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;
```

(continued on next page)

Example 2–10 (Cont.) Example of Data-Oriented TEXT_IO

```
   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;


         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;
```

(continued on next page)

Example 2–10 (Cont.) Example of Data-Oriented TEXT_IO

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

end ANOTHER_IO_EXAMPLE;
```

2.7.1.3 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.

Therefore, 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 ignores 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.

2.7.1.4 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 2–11 shows the use of this method. The example also shows how you can display a default value that is 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 2–11 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;

begin
```

(continued on next page)

Example 2–11 (Cont.) Example of Flexible TEXT_IO

```
   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)

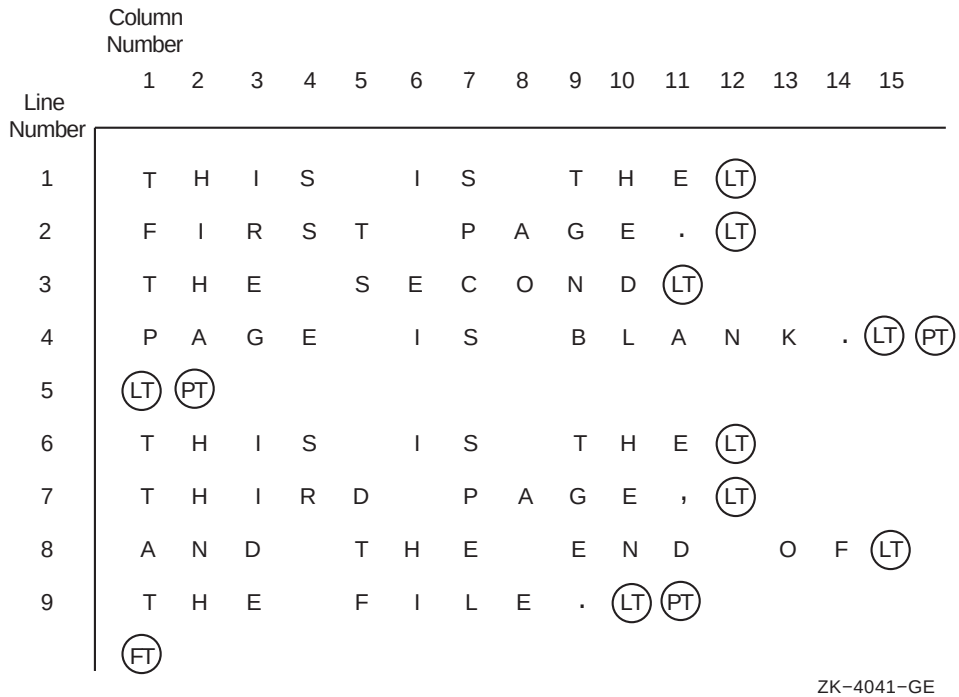
Example 2–11 (Cont.) Example of Flexible TEXT_IO

```
begin
   OPEN(INPUT,IN_FILE,"TT:");
   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;
```

2.7.2 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 *DEC 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 2–3 shows a three-page text file.

Figure 2–3 An Ada Text File, Showing Line, Page, and File Terminators



DEC Ada interprets these terminators as follows:

- A line terminator (LT) is designated by the end of an 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. In a line of indefinite line length, the line terminator occurs in the last record in the line, and has the same properties as any other line terminator. See Section 2.7 for more information about indefinite-line-length files.
- 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 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 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 RMS record in the corresponding external file. (See the *DEC 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).

2.7.3 Text Input-Output Buffering

Line buffering is done for most text input-output operations (terminal or nonterminal). Line buffering means that as characters are read or written to a DEC Ada text file, they are stored in an internal line buffer until a complete record can be transferred through RMS. Line buffering is done because DEC Ada TEXT_IO operations are implemented with RMS input-output operations. RMS operations always involve complete records, so the transfer of characters between a physical input-output device and a DEC Ada text file is complete only when a line terminator is detected (except in certain cases involving indefinite-line-length files).

Line buffering has the following effects:

- Terminal input is not processed until the line is terminated by a carriage return (or other line terminator).
- In situations when you provide more information in a line than the current input operation needs, 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.
- Text 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 2.3). (PRINT is the default CARRIAGE_CONTROL attribute provided by the package TEXT_IO for external files that are terminals; see Table 2–13.)

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.

2.7.4 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 2.3, 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 2.3.2), 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 RMS line termination operation occurs.
FORTRAN	Useful for applications that want to use FORTRAN carriage-control characters.

Table 2–14 summarizes the meaning of the FDL CARRIAGE_CONTROL values when they are applied to DEC Ada text files (for both terminal and nonterminal input-output).

Table 2–14 DEC 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).
	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.

(continued on next page)

Table 2–14 (Cont.) DEC Ada Carriage-Control Options

Option	Kind of Input-Output	Carriage Control
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.
	Terminal output Nonterminal output	An 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 DEC Ada. The first byte PUT by the user in each line is interpreted as a FORTRAN carriage-control character (see Table 2–15).

Table 2–15 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.
' 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.

2.7.5 Predefined Instantiations of TEXT_IO Packages

To make your use of the generic TEXT_IO operations more efficient, DEC 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[1]	FLOAT_IO(LONG_LONG_FLOAT)

[1]On VAX systems only.

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;
```

2.8 Input-Output and Exception Handling

The DEC Ada input-output packages raise errors that are defined in the packages IO_EXCEPTIONS and AUX_IO_EXCEPTIONS. See the *DEC Ada Language Reference Manual* for descriptions of these errors and the operations that raise them. See Chapter 3 of this manual for information on exception handling.

2.9 Input-Output and Tasking

DEC 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.

Also, each file operation is synchronized so that a series of operations to the same file takes place sequentially, rather than concurrently. Operations on the same file are indivisible. If one task is performing an operation on a file and another task attempts to perform an operation on the same file, DEC 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 lets multiple tasks 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". DEC Ada does not produce "ReOuacth".

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

3
Exception Handling

DEC Ada exception handling as defined in Chapter 11 of the *DEC Ada Language Reference Manual*, is implemented using the routines and related OpenVMS system services that comprise the OpenVMS condition handling facility. However, DEC Ada exception handling is implemented so that you do not need to call condition-handling facility routines and services directly. This chapter outlines how DEC Ada exception handling is related to OpenVMS 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 DEC Ada pragmas to import and export exceptions are covered in Chapters 11 and 13 of the *DEC Ada Language Reference Manual*. The condition-handling facility is described in the *OpenVMS Calling Standard*. You should be familiar with the material in these manuals before using the information in this chapter.

3.1 Relationship Between Ada Exception Handling and OpenVMS Condition Handling

All DEC Ada exceptions are encoded as OpenVMS 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 DEC Ada condition value denoted by the name ADA$_EXCEPTION plus a signal argument that is the address of the exception name.

Section 3.1.1 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 OpenVMS Run-Time Library routine 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 OpenVMS condition is propagated to an Ada program from the external environment.

In DEC 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—an OpenVMS or DEC 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 an OpenVMS 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 propagates to both of the tasks involved in the rendezvous.)
- If an unhandled OpenVMS 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 OpenVMS 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 2 for more information on how these names are interpreted. See the *OpenVMS Calling Standard* for more information about OpenVMS default handlers. See Chapter 7 for more information on exception handling and tasks.

Table 3–1 summarizes the DEC Ada implementation of exception handling.

Table 3–1 Relationship Between Ada Exception Handling and the OpenVMS Condition-Handling Facility

Ada Exception Handling	Condition-Handling-Facility Implementation
Enter an Ada frame with an exception part.[1]	Maintain information about currently active exception parts with a pointer to an internal DEC Ada run-time table. On VAX systems, establish the general DEC Ada condition handler for the surrounding stack frame.[2]
Raise an exception.	Signal a condition with a call to the LIB$STOP routine, or signal a hardware-generated condition.
Invoke an exception handler.	Unwind (SYS$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 the LIB$STOP routine with a copy of the signal arguments that caused invocation of the currently active exception part. The SS$_RESIGNAL feature of the condition-handling facility is not used to re-raise an exception.
No handler for the exception.	Signal a condition with a call to the LIB$SIGNAL routine (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.

[2]The term *stack frame* (synonymous with the term *call frame*) refers to a run-time OpenVMS 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 3–1 (Cont.) Relationship Between Ada Exception Handling and the OpenVMS Condition-Handling Facility

Ada Exception Handling	Condition-Handling-Facility Implementation
Raise an Ada format exception.[3]	Call the LIB$STOP routine with the condition value ADA$_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 OpenVMS format exception.[3]	Call the LIB$STOP routine with a unique 32-bit OpenVMS condition value.

[3]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 OpenVMS format are any predefined exceptions or any user-defined exceptions declared with import-export pragmas that specify a value of OpenVMS for the pragma FORM parameter. See Sections Section 3.1.1, Section 3.4.1,
and Section 3.4.2.

The raising of an exception in an Ada program involves calling the LIB$STOP routine. (See Table 3–1.) 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 Section 3.4.4 and Section 3.4.5 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 3.1.2 describes how signal argument copying is done and outlines some side effects.

3.1.1 Naming and Encoding Ada Exceptions

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

```
ADA$_exception_name
```

The exception PROGRAM_ERROR (from the package STANDARD) has the symbolic name ADA$_PROGRAM_ERROR, the exception DATA_ERROR (from the package IO_EXCEPTIONS) has the symbolic name ADA$_DATA_ERROR, and so on.

The predefined exceptions are listed in Table 3–2. The situations in which they are raised are described in the *DEC Ada Language Reference Manual*.

Table 3–2 Ada Predefined Exceptions

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

Ada lets you 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 lets you 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 OpenVMS condition value with the symbolic name ADA$_EXCEPTION, plus an additional signal argument that makes each value unique. This signal argument is the address 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 OpenVMS format by using the pragmas IMPORT_EXCEPTION and EXPORT_EXCEPTION. See Section 3.4 for more information.

3.1.2 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 DEC 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 DEC 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 DEC 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 DEC 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 is not lost in the following cases:

- For Ada exceptions
- For non-Ada conditions that have no FAO arguments
- For hardware conditions
- For RMS conditions
- For OpenVMS 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).

3.1.3 The Matching of Ada Exceptions and System-Defined Conditions

In DEC 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 OpenVMS condition (see Section 3.4.1), it matches an exception choice only if the name in the exception choice matches the internal name of the imported OpenVMS condition. Imported OpenVMS conditions may also match the exception choice **others** and the exception choice SYSTEM.NON_ADA_ERROR (see Section 3.4.3).

Some OpenVMS conditions are treated as being equivalent to certain Ada predefined exceptions. See Table 3–3 for a listing. 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. These conditions do not match the exception choice SYSTEM.NON_ADA_ ERROR. See Section 3.4.3.

Table 3–3 System-Defined Conditions that Match Ada Exceptions

Condition Name	Meaning	Exception Name
CMA$E_STACKOVF[1]	DECthreads stack overflow	STORAGE_ERROR
CMA$E_NOSTACKMEM[1]	DECthreads no stack memory	STORAGE_ERROR
CMA$E_INSFMEM[1]	DECthreads insufficient memory	STORAGE_ERROR
MTH$_UNDEXP[1]	Undefined exponentiation	CONSTRAINT_ERROR
SS$_FLTDIV	Floating/decimal divide by zero trap	CONSTRAINT_ERROR
SS$_FLTDIV_F	Floating divide by zero fault	CONSTRAINT_ERROR
SS$_FLTOVF	Floating overflow trap	CONSTRAINT_ERROR
SS$_FLTOVF_F	Floating overflow fault	CONSTRAINT_ERROR
SS$_HPARITH[1]	High-performance arithmetic trap	CONSTRAINT_ERROR
SS$_INTDIV	Integer divide by zero trap	CONSTRAINT_ERROR
SS$_INTOVF	Integer overflow trap	CONSTRAINT_ERROR
SS$_RANGEERR[1]	Range error	CONSTRAINT_ERROR
SS$_STKOVF	Stack overflow	STORAGE_ERROR

[1]On Alpha systems only.

These conditions match CONSTRAINT_ERROR or STORAGE_ERROR but are not converted to CONSTRAINT_ERROR or STORAGE_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 or STORAGE_ERROR (that is, you could have caught it with a CONSTRAINT_ERROR or STORAGE_ERROR handler), as well as the error message and traceback associated with the actual condition.

Note

On Alpha systems, the hardware access violations (SS$_ACCVIOs) that result during stack checking are converted to the condition SS$_STKOVF (which matches the Ada predefined exception STORAGE_ERROR. See Section 3.1.3 and Table 3–3.

3.2 Making the Best Use of Ada Exception Handling

To make the best use of Ada exception handling, keep the following 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 IS also invoked for any unexpected exception, such as "disk quota exceeded." A better solution would be to provide a specific handler for END_ERROR and let unexpected exceptions propagate, thereby making them visible.

Similarly, the following construct absorbs all exceptions without letting them 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 is re-raised in the containing frame.

3.3 Suppressing Checks

In accordance with the language definition, DEC Ada provides a set of run-time checks that underlie the predefined exceptions. The *DEC Ada Language Reference Manual* explains each check that the compiler performs and gives the corresponding exceptions that can arise. Table 3–4 summarizes this correspondence.

Table 3–4 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

To suppress checks, you can use the pragmas SUPPRESS and SUPPRESS_ALL (see the *DEC 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 OpenVMS 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 suppress INDEX_CHECK, but array processing continues whether or not you exceed the specified ranges. The results of that unit or subunit are unpredictable. If you are using the pragmas SUPPRESS or SUPPRESS_ALL or the /NOCHECK qualifier to improve the run-time performance of your program, consider using the techniques for eliminating checks discussed in Chapter 8.

The presence of the pragmas SUPPRESS or SUPPRESS_ALL or the use of the /NOCHECK qualifier does not guarantee that exceptions are not raised. For example, certain checks are not suppressed in DEC 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 some stack checks on VAX systems). 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 3–1.

3.4 Mixed-Language Exception Handling

DEC Ada provides the pragmas IMPORT_EXCEPTION and EXPORT_ EXCEPTION for use in a mixed-language programming environment:

- The pragma IMPORT_EXCEPTION lets you import an exception declared in a non-Ada program and handle it within your Ada program.
- The pragma EXPORT_EXCEPTION lets you export an Ada exception so that it can be treated as a OpenVMS condition in a non-Ada program.

DEC Ada also provides specific facilities for signaling and handling OpenVMS conditions in the OpenVMS 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 OpenVMS system routines, see Chapter 5.

3.4.1 Importing Exceptions

The pragma IMPORT_EXCEPTION associates an Ada exception name with either a OpenVMS condition value or another Ada exception name—both external to your program—and then lets you use that name in your Ada program. For example:

```
SQRT_NEGATIVE: exception;
pragma IMPORT_EXCEPTION (SQRT_NEGATIVE, "MTH$_SQUROONEG");
```

The full syntax and usage rules for this pragma are given in Chapter 13 of the *DEC Ada Language Reference Manual*.

Example 3–1 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;
```

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 OpenVMS 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 3.1.1 for definitions of Ada and OpenVMS formats. The format for imported exceptions is OpenVMS 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 OpenVMS condition MTH$_SQUROONEG; the pragma IMPORT_EXCEPTION associates the exception name SQRT_NEGATIVE with the OpenVMS condition MTH$_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 DEC 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 designator.
      "MTH$_SQUROONEG");  -- External designator.
                          -- By default use
                          -- OpenVMS format.

begin

   PUT_LINE("Get A, B, and C: ");
   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 OpenVMS condition SS$_NOPRIV 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 is used. (The VMS format is the required format for importing OpenVMS conditions.)

```
SS_NOPRIV: exception;
pragma IMPORT_EXCEPTION (
   SS_NOPRIV,        -- Internal designator.
   CODE => 16#24#);  -- Numeric condition value.
```

You can give a OpenVMS condition value to an Ada exception by first defining a message symbol condition with the OpenVMS Message Utility (see the *OpenVMS Command Definition, Librarian, and Message Utilities 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 designator.
   "MSG$_ERRORS",    -- External designator of a message symbol
                     -- defined with the OpenVMS Message Utility.
   FORM => VMS);     -- Explicitly specify VMS format.
```

Sections Section 3.4.4 and Section 3.4.5 provide additional discussion and examples of importing and handling OpenVMS conditions in the OpenVMS environment.

3.4.2 Exporting Exceptions

The pragma EXPORT_EXCEPTION associates an Ada exception with either a OpenVMS condition value or another Ada exception name and then lets you use that name in the external environment. For example:

```
ADA_ERROR : exception;
pragma EXPORT_EXCEPTION
   (ADA_ERROR,             -- Internal designator.
    "MY_PACKAGE_ADA",      -- External designator.
    FORM => ADA);          -- Ada format.
```

```
VMS_ERROR : exception;
pragma EXPORT_EXCEPTION
   (VMS_ERROR,             -- Internal designator.
    "MY_PACKAGE_ADA",      -- External designator.
    FORM => VMS,           -- VMS format.
    CODE => 16#8018004#);  -- VMS condition value.
```

The full syntax and usage rules for this pragma are given in Chapter 13 of the *DEC Ada Language Reference Manual*.

If you export a VMS format exception (see Section 3.1.1) to a non-Ada routine, that routine can treat the exception as an ordinary OpenVMS condition. In other words, the routine can process the exception using its own condition-handling mechanisms.

If you export an Ada format exception (see Section 3.1.1), 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.

3.4.3 The Exception Choice NON_ADA_ERROR

To let you treat non-Ada conditions as a special subclass of Ada exceptions, DEC Ada provides the exception choice NON_ADA_ERROR in the package SYSTEM. This exception choice is encoded as a predefined exception with the unique condition value ADA$_NON_ADA_ERROR, and it matches the following:

- Itself
- Any OpenVMS condition whose facility field is not ADA and which does not match an Ada exception (see Section 3.1.3)
- Ada exceptions for which the pragma IMPORT_EXCEPTION or EXPORT_EXCEPTION is given and for which the VMS format has been specified

For example:

```
with SYSTEM;
procedure SHOW_NON_ADA_ERROR is
   NEW_VMS: exception;
   pragma IMPORT_EXCEPTION(NEW_VMS, "SS$_EXQUOTA", VMS);
begin
   -- Some statements.
   exception
  when SYSTEM.NON_ADA_ERROR =>
     -- Handler that will catch NEW_VMS
     -- (plus other VMS conditions that qualify).
     end SHOW_NON_ADA_ERROR;
```

3.4.4 Signaling OpenVMS Conditions

DEC Ada provides two ways to signal an OpenVMS 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 OpenVMS Run-Time Library routine LIB$SIGNAL or LIB$STOP. The DEC 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. However, regardless of the condition's severity, the signal is noncontinuable.

For example, the Ada subprogram in Example 3–2 calls the system service SYS$GETJPIW, which returns information about one or more OpenVMS processes. When the condition SS$_NONEXPR occurs in Example 3–2, the SYS$GETJPIW routine returns the appropriate warning status. However, because the SS$_NONEXPR condition is imported and treated as an Ada exception, its severity becomes severe. Because an Ada exception handler exists for the exception NONEXPR, control passes to the handler.

Example 3–2 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));

   -- Declare the possible errors as Ada exceptions.
   --
    . . .
   NONEXPR: exception;
   pragma IMPORT_EXCEPTION(NONEXPR,"SS$_NONEXPR");
   NOPRIV : exception;
   pragma IMPORT_EXCEPTION(NOPRIV, "SS$_NOPRIV");
    . . .
```

(continued on next page)

Example 3–2 (Cont.) Handling SYS$GETJPIW Status Values as Ada Exceptions

```
   -- 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;


-- Handle the exceptions.
--
exception
    . . .
   when NONEXPR =>
      PUT_LINE("Nonexistent process");
   when NOPRIV  =>
      PUT_LINE("Insufficient privileges");
    . . .

end GETJPI_ADA;
```

(continued on next page)

Example 3–2 (Cont.) Handling SYS$GETJPIW Status Values as Ada Exceptions

When you signal a condition using the CONDITION_HANDLING.SIGNAL procedure (or another Ada equivalent to the OpenVMS Run-Time Library LIB$SIGNAL routine), the condition behaves like a OpenVMS condition, according to condition-handling-facility rules. If the condition's severity is not severe (error, warning, or informational), then the signal is continuable. Example 3–3 rewrites Example 3–2 to achieve this effect. In Example 3–3, CONDITION_HANDLING.SIGNAL is used so that the warning status of the NONEXPR condition is preserved and continuation occurs.

Example 3–3 Handling SYS$GETJPIW Status Values as OpenVMS 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_OpenVMS 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;
     . . .
```

(continued on next page)

Example 3–3 (Cont.) Handling SYS$GETJPIW Status Values as OpenVMS Conditions

```
    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;

begin

   -- Call SYS$GETJPIW using the interface from the package STARLET.
   --
   GETJPIW(STATUS => JPI_STATUS,    PIDADR => PIDADR,
           ITMLST => JPI_ITEM_LIST, IOSB   => IOSB_VALUE);
```

(continued on next page)

Example 3–3 (Cont.) Handling SYS$GETJPIW Status Values as OpenVMS Conditions

```
   -- 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_OpenVMS;
```

If, in Example 3–3, you were to use the CONDITION_HANDLING.STOP procedure (or another Ada equivalent to the OpenVMS 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 OpenVMS condition that is raised becomes severe and is beyond the control of the code that originally signaled it.

3.4.5 Effects of Handling OpenVMS Conditions from an Ada Program

If an Ada subprogram handles a OpenVMS 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 OpenVMS Run-Time Library fault handlers in a program that calls Ada subprograms (see Section 3.4.6), 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 condition-handling facility when a OpenVMS condition is signaled.
- An OpenVMS condition that is handled by an Ada handler is converted to a noncontinuable exception.

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 OpenVMS conditions:

- The exception choice **others**—catches any OpenVMS condition or Ada exception that does not have an explicit handler
- The DEC Ada predefined exception SYSTEM.NON_ADA_ERROR (see Section 3.4.3)
- Predefined Ada exceptions that match OpenVMS conditions (see Section 3.1.3)

Note

The exception choice **others** does not catch the following OpenVMS conditions:

SS$_DEBUG
SS$_UNWIND

These two OpenVMS conditions are used to implement the DEC 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:

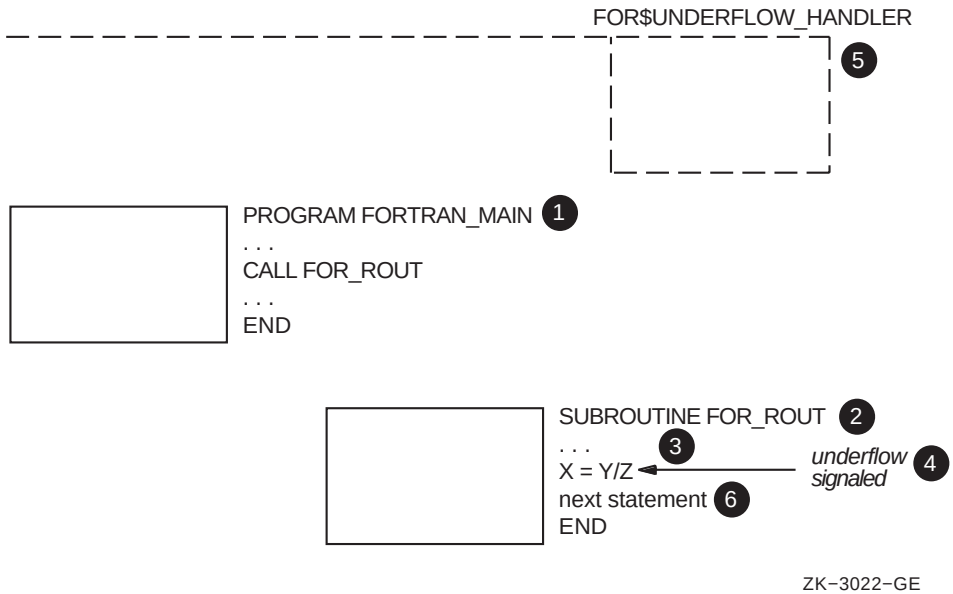
- A Fortran program P1_FOR calls an Ada subprogram P2_ADA.
- The Ada subprogram calls another Fortran program, P3_FOR.

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 Fortran documentation for more information). 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 3–1 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.

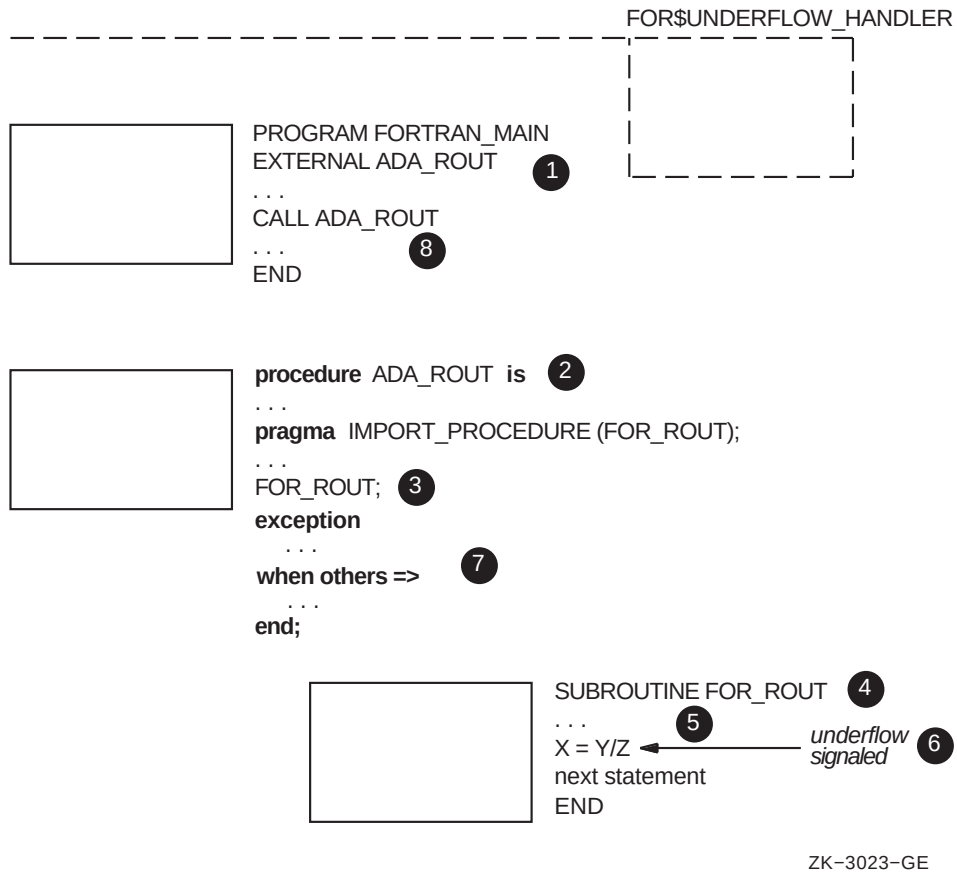
Figure 3–1 Execution of a Fortran Program with FOR$UNDERFLOW_HANDLER



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 3–2 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 3–2 The Effect of an Ada Procedure Containing an Others Handler



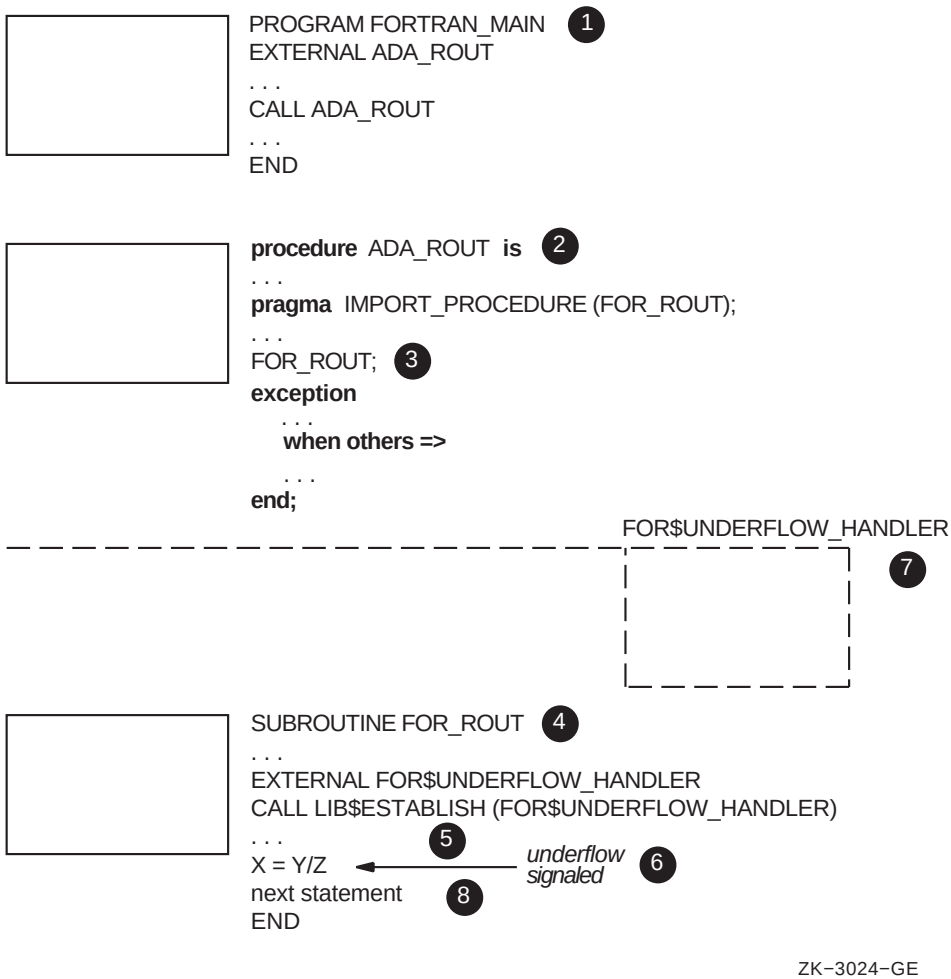
There are two flaws in this example:

- 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 OpenVMS condition it never intended to catch

One solution for improving this example is to use OpenVMS Run-Time Library calls to explicitly establish FOR$UNDERFLOW_HANDLER in the places where you want to handle underflow. Figure 3–3 shows a repaired version of the Fortran program, which instead establishes FOR$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.

Figure 3–3 FOR$UNDERFLOW_HANDLER Established for a Fortran Subroutine



ZK–3024–GE

3.4.6 Fault Handlers (VAX Systems Only)

Fault handlers are usually established as catch-all stack handlers, or they are called from condition handlers (such as the OpenVMS 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 DEC 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 3.1.2 are accessible only to the Ada run-time library and not to the underlying condition-handling-facility routines). You cannot do the following:

- Establish a fault handler in Ada.
- Call a OpenVMS 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 OpenVMS 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 condition-handling facility 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.

See Chapter 5 of this manual for information on how to call system services and see the *OpenVMS Calling Standard* for an explanation of exception vectors.

3.5 Exceptions and Tasking

Exception handling in DEC 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 DEC Ada program, the DEC Ada run-time library displays the exception message to warn you that the task is terminating. You can also use the debugger to diagnose this situation (see *Developing Ada Programs on OpenVMS 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 causes 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 lets a reader of your code 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 eventually reaches 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 7.4.6 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 debugger to diagnose situations in which an exception may wait forever for a dependent task (see *Developing Ada Programs on OpenVMS Systems*).

Third, DEC 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 OpenVMS DCL command interpreter. This additional DEC 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 OpenVMS default handler (see Section 3.1). In cases where such a task fails to terminate, you

can use the debugger to diagnose the problem (see *Developing Ada Programs on OpenVMS 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, DEC 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 3.1.2). 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).

4

Mixed-Language Programming

To write mixed-language programs, you need to be able to pass parameters, return function results, and share data in spite of differences in conventions and features between languages. DEC Ada implements a number of features designed to simplify other-language calls to and from Ada programs.

This chapter explains how to accomplish mixed-language programming with DEC Ada. You should be familiar with the following information before using the information in this chapter:

- Chapter 6 of the *DEC Ada Language Reference Manual*. This chapter gives the Ada semantics rules for subprogram calls.
- Chapter 13 of the *DEC Ada Language Reference Manual*. This chapter provides information on Ada features and pragmas for controlling data representation, as well as the detailed syntax and usage rules for the DEC Ada pragmas that support mixed-language programming.
- Chapter 1 of this manual. This chapter provides specific information on data representation in DEC Ada.

You should also be familiar with the calling standard that defines the multilanguage environment on your system. See the *OpenVMS Calling Standard* for more information.

The calling of system routines and other callable utilities and tools is a specific example of mixed-language programming. See Chapter 5 for more information on that topic.

Note

This manual uses the term *subprogram* to refer to Ada procedures and functions. It uses the term *external routine* or *routine* to refer to other-language code that calls or is called by Ada subprograms in mixed-language programs.

4.1 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 is not part of the Ada program, you must give the predefined pragma INTERFACE with the Ada specification. For example:

```
function C_ADD(X: INTEGER; Y: INTEGER) return INTEGER;
pragma INTERFACE(C, C_ADD);
```

The syntax and rules for using the pragma INTERFACE are described in detail in Chapter 13 of the *DEC Ada Language Reference Manual*.

When you import a routine and specify ADA, C, FORTRAN, BLISS, or DEFAULT as the language name in the pragma INTERFACE, the DEC Ada compiler chooses an appropriate set of default passing mechanisms for any parameters or function results involved. So, you can import a routine written in one of these languages without having to know and explicitly specify the passing mechanisms required by the other-language routine.

Note

In some cases, default mechanisms for passing parameters or for returning function results are not chosen. See Section 4.3.2 for more information.

If you specify a language name other than ADA, FORTRAN, C, BLISS, or DEFAULT, DEC Ada chooses the conventions associated with the language name DEFAULT. You can use the language name DEFAULT to specify an interface to a language that follows the OpenVMS calling standard.

If you want to know which mechanisms the compiler has chosen, use the /WARNING=COMPILATION_NOTES qualifier with any of the Ada compilation commands. For example:

```
$ ADA/WARNINGS=COMPILATION_NOTES ADD

    6       function C_ADD(X: INTEGER; Y: INTEGER) return INTEGER;
.....................................................1
%ADAC-I-PASS_MECH_IS, (1) Selected or specified passing mechanism is VALUE
.......................2
%ADAC-I-PASS_MECH_IS, (2) Selected or specified passing mechanism is VALUE
...................................3
%ADAC-I-PASS_MECH_IS, (3) Selected or specified passing mechanism is VALUE
```

For example, the following Ada program imports a routine written in C:

```
with TEXT_IO; use TEXT_IO;
with C_TYPES; use C_TYPES;
procedure ADD is
   A: INT := 10;
   B: INT := 539;
   function C_ADD(X: INT; Y: INT) return INT;
   pragma INTERFACE(C, C_ADD);
   package INT_TEXT_IO is new INTEGER_IO(INT);
   use INT_TEXT_IO;
begin
   PUT("A + B is ");
   PUT(C_ADD(A,B));
   NEW_LINE;
end ADD;

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

int c_add (int x, int y)
   {
   return x+y;
   }
```

Once you have coded the Ada specification and the routine to be imported, you compile them using the appropriate compilers. Then, you link and run them using the appropriate commands. For example:

```
$ ADA ADD
$ CC C_ADD
$ ACS LINK ADD C_ADD
$ RUN ADD
```

See *Developing Ada Programs on OpenVMS Systems* for more information on linking mixed-language programs.

In some cases, you may want to specify more information about the imported routine. In those cases, you specify one of the following pragmas in addition to the pragma INTERFACE:

- The pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE allow you to partially or fully specify all of the parameter-passing and function result mechanisms.
- The pragmas INTERFACE_NAME, IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE allow you to explicitly designate the name of the external routine to be imported.
- The pragma IMPORT_VALUED_PROCEDURE allows you to establish that your imported routine is a valued procedure: like a function, it returns a result value; like a procedure, it can also cause side effects on its parameters.

The syntax and rules for using the DEC Ada import pragmas are described in detail in Chapter 13 of the *DEC Ada Language Reference Manual*.

For example, the following Ada program imports the C routine as in the preceding example, but in this case the Ada program explicitly specifies the VALUE mechanism for passing the parameters and the function result:

```
with TEXT_IO; use TEXT_IO;
with C_TYPES; use C_TYPES;
procedure ADD is
   A: INT := 10;
   B: INT := 539;
   function C_ADD (X: INT; Y: INT) return INT;
   pragma INTERFACE (C, C_ADD);
   pragma IMPORT_FUNCTION (INTERNAL        => C_ADD,
                           MECHANISM       => (VALUE, VALUE),
                           RESULT_MECHANISM => VALUE);
   package INT_TEXT_IO is new INTEGER_IO(INT);
   use INT_TEXT_IO;
begin
   PUT("A + B is ");
   PUT(C_ADD(A,B));
   NEW_LINE;
end ADD;

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

int c_add (int x, int y)
   {
   return x+y;
   }
```

See Section 4.3 for more information on controlling parameter-passing and function result mechanisms.

When importing external routines, note the following:

- The Ada language does not allow you to pass subprograms as parameters, except to generic instantiations (see Chapter 12 of the *DEC Ada Language Reference Manual*). You can, however, pass the address of a subprogram by first exporting it and then taking its address with the 'ADDRESS attribute. The 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 imported or exported, the compiler issues a warning message and uses the value SYSTEM.ADDRESS_ZERO as the result.

  See Section 4.2 for more information about exporting subprograms. See Section 9.1 for more information about taking the address of a subprogram.

- DEC Ada allows you to specify an imported Ada routine in an address clause. See Example 4–1.

- The Ada language allows you to pass task entries as parameters only to generic instantiations (see Chapter 12 of the *DEC Ada Language Reference Manual*).

When passing parameters or returning results of certain types (for example, array or record types), some more complex Ada conventions may apply. See Appendix B for more information.

Example 4–1 Using an Address Clause to Make Indirect Calls

```
-- This example shows the use of address clauses with imported
-- subprograms.
--
-- The program declares a record type, and defines a PUT
-- procedure for printing the values of the record components.
--
-- The PUT procedure definition demonstrates the use of
-- an address clause. The address clause is used to
-- index into a table of subprograms.
--
package ADDRESS_PKG is
```

(continued on next page)

Example 4–1 (Cont.) Using an Address Clause to Make Indirect Calls

```
   -- Define the set of record types.
   --
   type KIND is (K1, K2, K3);
   type NODE(K : KIND) is
      record
         case K is
            when K1 => C1 : INTEGER;
            when K2 => C2 : CHARACTER;
            when K3 => C3 : BOOLEAN;
         end case;
      end record;

   -- Declare the PUT procedure for printing out
   -- NODE values.
   --
   procedure PUT(N : NODE);

end ADDRESS_PKG;

with SYSTEM;
with TEXT_IO; use TEXT_IO;
package body ADDRESS_PKG is

   -- Declare a set of PUT procedures, one for each component
   -- of the NODE record type. These are to be called by the
   -- general PUT procedure.
   --
   -- Each separate procedure is exported, causing the compiler
   -- to generate code that uses the standard calling
   -- conventions, and thus making it possible to take the
   -- address of each procedure.
   --
   procedure PUT_K1(N : NODE) is
   begin
      PUT_LINE(INTEGER'IMAGE(N.C1));
   end PUT_K1;
   pragma EXPORT_PROCEDURE(PUT_K1, EXTERNAL=>"");

   procedure PUT_K2(N : NODE) is
   begin
      PUT_LINE(CHARACTER'IMAGE(N.C2));
   end PUT_K2;
   pragma EXPORT_PROCEDURE(PUT_K2, EXTERNAL=>"");
```

(continued on next page)

Example 4–1 (Cont.) Using an Address Clause to Make Indirect Calls

```
   procedure PUT_K3(N : NODE) is
   begin
      PUT_LINE(BOOLEAN'IMAGE(N.C3));
   end PUT_K3;
   pragma EXPORT_PROCEDURE(PUT_K3, EXTERNAL=>"");

   -- Declare a table of the addresses for each of the
   -- separate PUT procedures.
   --
   package TABLE is
      ADDRESSES : constant array(KIND) of
                  SYSTEM.ADDRESS := (K1 => PUT_K1'ADDRESS,
                                     K2 => PUT_K2'ADDRESS,
                                     K3 => PUT_K3'ADDRESS);
   end TABLE;

   -- Implement the general PUT procedure. This procedure
   -- declares a procedure PUT_K. An address clause associates
   -- PUT_K with the appropriate procedure in the table of
   -- procedure addresses, so that PUT_K can represent any of
   -- the exported procedures.
   --
   procedure PUT(N : NODE) is
      procedure PUT_K(N : NODE);
      pragma INTERFACE(ADA, PUT_K);
      pragma IMPORT_PROCEDURE(PUT_K, EXTERNAL => "");
      for PUT_K use at TABLE.ADDRESSES(N.K);
   begin
      PUT_K(N);
   end PUT;

end ADDRESS_PKG;

with ADDRESS_PKG; use ADDRESS_PKG;
procedure ADDRESS_CLAUSE_EXAMPLE is
begin
   PUT((K1, 1));
   PUT((K2, '2'));
   PUT((K3, FALSE));
end ADDRESS_CLAUSE_EXAMPLE;
```

4.2 Calling Ada Subprograms from External Routines

You call an Ada subprogram from an external routine by first writing the specification and body for the Ada subprogram. To indicate that the subprogram will be called from an external routine, you must also give

one of the DEC Ada export pragmas: EXPORT_FUNCTION, EXPORT_PROCEDURE, or EXPORT_VALUED_PROCEDURE.

The syntax and rules for using the DEC Ada export pragmas are described in detail in Chapter 13 of the *DEC Ada Language Reference Manual*.

For example, the following Ada procedure is called by a C routine:

```
procedure SQUARE (NUM : in INTEGER; RESULT : out INTEGER) is
begin
   RESULT := NUM * NUM;
end SQUARE;
pragma EXPORT_PROCEDURE (INTERNAL => SQUARE,
                         MECHANISM => (VALUE, REFERENCE));
```

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

extern void square (
   int num, int*result)

main()
{
   int res;

/*
 * Call Ada procedure.
 */
   {
   square (10,&res);
   printf ("Result: %D\n",res);
   };
}
```

Once you have coded the Ada subprogram and the calling routine, compile them using the appropriate compilers. Then, link and run them using the appropriate commands. For example:

```
$ ADA SQUARE
$ CC MAIN
$ ACS EXPORT SQUARE
$ LINK MAIN,SQUARE
```

See *Developing Ada Programs on OpenVMS 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 with mechanisms that are acceptable to the Ada subprogram. You can do this by specifying the appropriate passing mechanisms with the pragma EXPORT_PROCEDURE, as shown in the preceding example.

When exporting a subprogram, you can use the compiler compilation notes messages to determine which mechanisms the DEC Ada compiler chooses for the parameters in an Ada subprogram. To obtain these messages, use the compiler qualifier /WARNINGS=COMPILATION_NOTES when you compile your Ada subprogram. For example:

```
$ ADA/WARNINGS=COMPILATION_NOTES SQUARE

    1   procedure SQUARE (NUM : in INTEGER; RESULT : out INTEGER) is
.........................1
%I, (1) Selected/specified passing mechanism is VALUE
.........................................2
%I, (2) Selected/specified passing mechanism is REFERENCE
```

See Section 4.3 for more information on controlling parameter-passing and function result mechanisms.

Note

Some programming languages allow optional and/or default parameters. Calls from external routines to exported DEC Ada subprograms must supply all parameters that are declared as formal parameters. In particular, you must supply an actual value, even when you give a default expression for a formal parameter in the Ada subprogram specification.

When passing parameters or returning function results of certain types (for example, array or record types), some more complex Ada conventions may apply. See Appendix B for more information.

4.3 Controlling the Passing Mechanisms for Imported and Exported Subprogram Parameters and Function Results

When importing or exporting other-language routines or exporting Ada subprograms, you may want to explicitly specify the passing mechanisms for one or more parameters or function results. For example:

- You may want to ensure that a particular mechanism is always chosen.
- You may be passing parameters to an imported routine for which the compiler does not choose a default mechanism (see Section 4.3.2).
- You may be importing a routine that is not written in C, Ada, Fortran, or BLISS.

- You may be exporting an Ada subprogram to another language that traditionally uses different default mechanisms (for example, Fortran generally passes parameters by reference by default).

Before you decide to specify explicitly the passing mechanisms for an imported or exported subprogram, you can use the compiler compilation notes to determine which default mechanisms the compiler chooses for certain parameters or function results. You obtain these notes by using the /WARNINGS=COMPILATION_NOTES qualifier on any of the compilation commands. See *Developing Ada Programs on OpenVMS Systems* for more information.

Note

The compiler chooses the same mechanisms for exported subprograms as it does for imported routines for which the language ADA is specified in the pragma INTERFACE.

For imported routines if you specify a language name other than ADA, FORTRAN, C, BLISS, or DEFAULT in the pragma INTERFACE, DEC Ada chooses the conventions associated with the language name DEFAULT.

4.3.1 Using the MECHANISM and RESULT_MECHANISM Options

Once you have decided to explicitly specify your parameter-passing mechanisms, you can use the MECHANISM option in the DEC Ada import or export pragmas to specify one of three values for each parameter. Similarly, you can use the RESULT_MECHANISM option to specify one of the same three values for each function result. The three mechanisms you can specify are as follows:

- VALUE—causes the value of the actual parameter or function result to be passed or returned.
- REFERENCE—causes an address of the value of the actual parameter to be passed; causes the address of the function result to be returned by the extra parameter method (see Section 4.4.2).
- DESCRIPTOR (optionally including the descriptor class)—causes the address of a string, array, or scalar descriptor to be passed or returned. Function results are returned by the extra parameter method (see Section 4.4.2). See Appendix B for more information about descriptors and descriptor classes.

Note

Passing the address of a value with the VALUE mechanism is not the same as passing the value by the REFERENCE mechanism. In some instances, the REFERENCE mechanism may involve passing the address of a copy of the value.

In addition to the mechanism chosen for a particular parameter, the Ada parameter-passing semantics and DEC Ada linkage conventions also apply. The Ada semantics are determined by the parameter's type. See Section 4.4.1 for more information on Ada semantics; see Section 4.4.2 for more information on the DEC Ada linkage conventions.

Table 4–1 gives the usage rules for the MECHANISM options. Table 4–2 gives the usage rules for the RESULT_MECHANISM options. See Sections 4.6 and 4.7 for examples in which parameter and result mechanisms are specified.

Table 4–1 Parameter-Passing Mechanisms and Allowed Data Types

MECHANISM Option	Allowed Ada Types	Other Usage Rules
VALUE	All except unconstrained arrays and records	On VAX systems, the type or subtype must have a compile-time size of 32 bits or less. On Alpha systems, the type or subtype must have a compile-time size of 64 bits or less. When you apply this mechanism to the first parameter of a valued procedure (a procedure specified with the pragma IMPORT_VALUED_PROCEDURE or EXPORT_VALUED_PROCEDURE), the type or subtype is treated as a function result. See Table 4–2 for more information. Parameters must be of mode **in** or they must be the first parameter of a valued procedure. First parameters of valued procedures must be of mode **out**.
REFERENCE	All	For exported subprograms or imported subprograms where the language is ADA, array parameters must be constrained and byte-aligned. See Appendix B for information on passing types with discriminants with defaults.
DESCRIPTOR	All except task or record types; not allowed for scalar types in exported subprograms	See Appendix B.

Table 4–2 Function Return Mechanisms and Allowed Data Types

RESULT_MECHANISM Option	Allowed Ada Types	Other Usage Rules[1]
VALUE	All except unconstrained arrays	Type or subtype must have a compile-time size of 64 bits or less. For exported functions or imported functions where the language is ADA, array results must be constrained.
REFERENCE	All except unconstrained record or array types	The result is returned by the extra parameter method (see Section 4.4.2).
DESCRIPTOR	All except task or record types	The result is returned by the extra parameter method (see Section 4.4.2). For unconstrained array and record results, an area control block is used in addition to a descriptor. See Appendix B.

[1] For unconstrained strings, arrays, and records, use an area control block. See Appendix B.

If the allocation of an enumeration or integer type is less than the maximum allowed size, the returned result is zero- or sign-extended (as appropriate) to a clean representation.

The calling routine must use the extra parameter method (see Section 4.4.2) for an array type result if the storage size is not known at compile time, or if the size is known to be greater than 64 bits. The calling routine must also use the extra parameter method for a record type result if the storage size is greater than 64 bits.

You can use the REFERENCE mechanism even in cases where a descriptor or additional information may be passed. 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 DEC Ada you would normally pass an unconstrained array parameter by descriptor, where the descriptor contains the information about the array bounds. 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 = 0.0
      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 DEC Ada would be as follows:

```
procedure CALL_INNERPROD is

   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;

begin
    . . .
   P := INNERPROD (Q, T, Q'LENGTH);
    . . .
end CALL_INNERPROD;
```

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.

4.3.2 Working with Imported Routine Parameters or Function Results for Which There Are No Defaults

When you import a routine, the Ada compiler chooses default passing mechanisms for parameters and function results based on the language specified in the associated pragma INTERFACE. In some cases, no defaults are chosen.

Table 4–3 lists the languages and data types for which no defaults are chosen for imported routine parameters. Table 4–4 lists the languages and data types for which no defaults are chosen for imported function results. In these languages and for parameters and function results of these types, you must specify the passing mechanisms. See Sections 4.6 and 4.7 for C and Fortran examples involving routine parameters for which mechanisms must be specified.

Note

The C language does not have the concept of **in out** or **out** parameters. Thus, no default mechanism is chosen for an **in out** or **out** parameter, regardless of its type, if the C language (or its equivalent) is specified for an imported routine.

Table 4–3 Cases in Which Mechanisms Must Be Specified for Imported Subprogram Parameters

Ada Type	Languages for Which There Are No Default Mechanisms[1]
The types STANDARD.LONG_FLOAT, SYSTEM.G_FLOAT, and SYSTEM.D_FLOAT[2]	C
in out or **out** parameters	C
Record	C
Task	C FORTRAN
Private	Depends on actual type

[1]Information that applies to the C language also applies to BLISS.

[2]On VAX systems only.

Table 4–4 Cases in Which Mechanisms Must Be Specified for Imported Function Results

Ada Type	Languages for Which There Are No Default Result Mechanisms[1]
String	C, Fortran
Array	C FORTRAN[2]
Record (constrained)	FORTRAN[2]
Record (unconstrained)	C FORTRAN[2]

[1]Information that applies to the C language also applies to BLISS.

[2]FORTRAN does not allow the return of arrays or records.

4.3.3 DEC Ada Equivalents for OpenVMS Data Types

For comparison and reference, Table 4–5 lists the Ada type equivalents and mechanisms supported for data types defined in the *OpenVMS Calling Standard*.

Table 4–5 DEC Ada Equivalents for OpenVMS Data Types and Their Valid Passing Mechanisms in DEC Ada

Data Type	Symbolic Code	DEC 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	Value[1], Reference[2], Descriptor[3]
Bound label value	DSC$K_DTYPE_BLV	Not available	—
Bound procedure value	DSC$K_DTYPE_ BPV	Not available	—

[1]The default for imported subprograms when the language specified in the pragma INTERFACE is C or BLISS.

[2]The default for imported subprograms when the language specified in the pragma INTERFACE is ADA, DEFAULT, or FORTRAN. Also the default or exported subprograms.

[3]Only when specified as a MECHANISM option of an import pragma.

(continued on next page)

Table 4–5 (Cont.) DEC Ada Equivalents for OpenVMS Data Types and Their Valid Passing Mechanisms in DEC Ada

Data Type	Symbolic Code	DEC Ada Translation	Passing Mechanism
Byte unsigned	DSC$K_DTYPE_BU	Any enumerated type whose values fit into an unsigned byte; SYSTEM.UNSIGNED_BYTE	Value[1], Reference[2], Descriptor[3]
COBOL intermediate temporary	DSC$K_DTYPE_CIT	Not available	—
D_floating	DSC$K_DTYPE_D	SYSTEM.D_FLOAT; LONG_FLOAT if pragmas FLOAT_REPRESENTATION (VAX_FLOAT) and LONG_FLOAT(D_FLOAT) are in effect	Value[1], Reference[2], Descriptor[3]
D_floating complex	DSC$K_DTYPE_DC	Not available[4]	See Tables Table 4–1 and Table 4–2
Descriptor	DSC$K_DTYPE_DSC	Not available[4]	See Tables Table 4–1 and Table 4–2[5]
F_floating	DSC$K_DTYPE_F	SYSTEM.F_FLOAT; FLOAT if pragma FLOAT_REPRESENTATION (VAX_FLOAT) is in effect	Value[1], Reference[2], Descriptor[3]
F_floating complex	DSC$K_DTYPE_FC	Not available[4]	See Tables Table 4–1 and Table 4–2

[1]The default for imported subprograms when the language specified in the pragma INTERFACE is C or BLISS.

[2]The default for imported subprograms when the language specified in the pragma INTERFACE is ADA, DEFAULT, or FORTRAN. Also the default or exported subprograms.

[3]Only when specified as a MECHANISM option of an import pragma.

[4]Can be simulated in DEC Ada with a record type definition. Complex types cannot be returned directly on Alpha systems. See Example 4–7 for information on how to pass and return complex numbers to and from a Fortran program.

[5]The default for imported subprogram string parameters when the language specified in the pragma INTERFACE is FORTRAN.

(continued on next page)

Table 4–5 (Cont.) DEC Ada Equivalents for OpenVMS Data Types and Their Valid Passing Mechanisms in DEC Ada

Data Type	Symbolic Code	DEC Ada Translation	Passing Mechanism
G_floating	DSC$K_DTYPE_G	SYSTEM.G_FLOAT; LONG_FLOAT if pragmas FLOAT_ REPRESENTATION (VAX_FLOAT) and LONG_FLOAT(G_FLOAT) are in effect	Value[1], Reference[2], Descriptor[3]
G_floating complex	DSC$K_DTYPE_GC	Not available[4]	See Tables Table 4–1 and Table 4–2
H_floating[6]	DSC$K_DTYPE_H	LONG_LONG_FLOAT; SYSTEM.H_FLOAT	Reference[2], Descriptor[3]
H_floating complex[6]	DSC$K_DTYPE_HC	Not available[4]	See Tables Table 4–1 and Table 4–2
IEEE S floating[7]	DSC$K_DTYPE_FS	SYSTEM.IEEE_SINGLE_ FLOAT; FLOAT if pragma FLOAT_ REPRESENTATION (IEEE_FLOAT) is in effect	
IEEE S floating complex[7]	DSC$K_DTYPE_FSC	Not available[4]	
IEEE T floating[7]	DSC$K_DTYPE_FT	SYSTEM.IEEE_ DOUBLE_FLOAT; LONG_FLOAT if pragma FLOAT_ REPRESENTATION (IEEE_FLOAT) is in effect	

[1]The default for imported subprograms when the language specified in the pragma INTERFACE is C or BLISS.

[2]The default for imported subprograms when the language specified in the pragma INTERFACE is ADA, DEFAULT, or FORTRAN. Also the default or exported subprograms.

[3]Only when specified as a MECHANISM option of an import pragma.

[4]Can be simulated in DEC Ada with a record type definition. Complex types cannot be returned directly on Alpha systems. See Example 4–7 for information on how to pass and return complex numbers to and from a Fortran program.

[6]On VAX systems only.

[7]On Alpha systems only.

(continued on next page)

Table 4–5 (Cont.) DEC Ada Equivalents for OpenVMS Data Types and Their Valid Passing Mechanisms in DEC Ada

Data Type	Symbolic Code	DEC Ada Translation	Passing Mechanism
IEEE T floating complex[7]	DSC$K_DTYPE_FTC	Not available[4]	
Longword integer (signed)	DSC$K_DTYPE_L	INTEGER	Value[1], Reference[2], Descriptor[3]
Longword (unsigned)	DSC$K_DTYPE_LU	SYSTEM.UNSIGNED_ LONGWORD	Value[1], Reference[2], Descriptor[3]
Numeric string, left separate sign	DSC$K_DTYPE_NL	STRING	See Tables Table 4–1 and Table 4–2
Numeric string, left overpunched sign	DSC$K_DTYPE_ NLO	STRING	See Tables Table 4–1 and Table 4–2
Numeric string, right separate sign	DSC$K_DTYPE_NR	STRING	See Tables Table 4–1 and Table 4–2
Numeric string, right overpunched sign	DSC$K_DTYPE_ NRO	STRING	See Tables Table 4–1 and Table 4–2
Numeric string, unsigned	DSC$K_DTYPE_NU	STRING	See Tables Table 4–1 and Table 4–2
Numeric string, zoned sign	DSC$K_DTYPE_NZ	STRING	See Tables Table 4–1 and Table 4–2
Octaword integer (signed)	DSC$K_DTYPE_O	Not available[4]	See Tables Table 4–1 and Table 4–2

[1]The default for imported subprograms when the language specified in the pragma INTERFACE is C or BLISS.

[2]The default for imported subprograms when the language specified in the pragma INTERFACE is ADA, DEFAULT, or FORTRAN. Also the default or exported subprograms.

[3]Only when specified as a MECHANISM option of an import pragma.

[4]Can be simulated in DEC Ada with a record type definition. Complex types cannot be returned directly on Alpha systems. See Example 4–7 for information on how to pass and return complex numbers to and from a Fortran program.

[7]On Alpha systems only.

(continued on next page)

Table 4–5 (Cont.) DEC Ada Equivalents for OpenVMS Data Types and Their Valid Passing Mechanisms in DEC Ada

Data Type	Symbolic Code	DEC Ada Translation	Passing Mechanism
Octaword logical (unsigned)	DSC$K_DTYPE_OU	Not available[4]	See Tables Table 4–1 and Table 4–2
Packed decimal string	DSC$K_DTYPE_P	Not available[4]	See Tables Table 4–1 and Table 4–2
Quadword integer (signed)	DSC$K_DTYPE_Q	SYSTEM.UNSIGNED_ QUADWORD, but arithmetic operations are not available	Value[1], Reference[2], Descriptor[3]
Quadword (unsigned)	DSC$K_DTYPE_QU	SYSTEM.UNSIGNED_ QUADWORD, but arithmetic operations are not available	Value[1], Reference[2]
Character string	DSC$K_DTYPE_T	STRING	See Tables Table 4–1 and Table 4–2
Aligned bit string	DSC$K_DTYPE_V	Packed BOOLEAN array	See Tables Table 4–1 and Table 4–2
Varying character string	DSC$K_DTYPE_VT	Not available[4]	See Tables Table 4–1 and Table 4–2
Unaligned bit string	DSC$K_DTYPE_VU	Packed BOOLEAN array	See Tables Table 4–1 and Table 4–2
Word integer (signed)	DSC$K_DTYPE_W	SHORT_INTEGER	Value[1], Reference[2], Descriptor[3]

[1]The default for imported subprograms when the language specified in the pragma INTERFACE is C or BLISS.

[2]The default for imported subprograms when the language specified in the pragma INTERFACE is ADA, DEFAULT, or FORTRAN. Also the default or exported subprograms.

[3]Only when specified as a MECHANISM option of an import pragma.

[4]Can be simulated in DEC Ada with a record type definition. Complex types cannot be returned directly on Alpha systems. See Example 4–7 for information on how to pass and return complex numbers to and from a Fortran program.

(continued on next page)

Table 4–5 (Cont.) DEC Ada Equivalents for OpenVMS Data Types and Their Valid Passing Mechanisms in DEC Ada

Data Type	Symbolic Code	DEC Ada Translation	Passing Mechanism
Word (unsigned)	DSC$K_DTYPE_WU	Any enumerated type whose values fit into an unsigned word; SYSTEM.UNSIGNED_WORD	Value[1], Reference[2], Descriptor[3]
Unspecified	DSC$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 subprograms when the language specified in the pragma INTERFACE is C or BLISS.

[2]The default for imported subprograms when the language specified in the pragma INTERFACE is ADA, DEFAULT, or FORTRAN. Also the default or exported subprograms.

[3]Only when specified as a MECHANISM option of an import pragma.

4.4 Ada Conventions for Passing Parameters and Returning Function Results in Mixed-Language Programs

When data is passed between routines that are not written in the same programming language, the calling routine must pass the data in a form and to a location recognized by the routine being called.

In DEC 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 used by DEC Ada to implement subprogram calls
- Any hardware- or system-specific calling standard

The following sections discuss the first two of these conventions.

4.4.1 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.

Note

An Ada program is erroneous if it depends on which mechanism is chosen for a particular parameter (see Chapter 6 of the *DEC Ada Language Reference Manual*).

Reference semantics is not the same as passing by the REFERENCE mechanism (see Section 4.3.1).

The Ada language requires copy-in/copy-back semantics for scalar and access type parameters (see Chapter 6 of the *DEC Ada Language Reference Manual*). DEC Ada follows these requirements. DEC 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 DEC Ada compiler takes advantage of this flexibility, and uses either kind of semantics for parameters of these types.

Note

When the DEC 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.

DEC 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.

4.4.2 DEC Ada Linkage Conventions

Linkage conventions describe the implementation of subprogram calls. DEC Ada makes subprogram calls as follows:

- Parameters are passed in an argument list or in registers, as appropriate to the underlying hardware. However, the first parameters in procedures specified with the pragmas IMPORT_VALUED_PROCEDURE or EXPORT_VALUED_PROCEDURE are treated as function results.
- Function results are returned in registers or are passed as extra leading parameters in the argument list. An extra leading parameter is used when the function value is returned by the REFERENCE or DESCRIPTOR mechanism. In this case, the calling routine passes the extra parameter—an address—as the first argument in the argument list. The address can point either to the storage for the value or to a descriptor. If the calling program allocates a descriptor, the called function must allocate storage for the function value and update the contents of the descriptor.

  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 Appendix B.

4.5 Sharing Data with Non-Ada Routines

When you write mixed-language programs, you must make sure that the data that is passed between Ada subprograms and non-Ada routines is in the form expected on both sides. For example, when you import a Fortran routine involving common blocks, you must set up the analogous Ada data structures so that the data is laid out and aligned in the way that Fortran expects it to be laid out and aligned.

DEC Ada provides a number of features that let you control the way data is formatted. In addition, it provides a set of pragmas designed for sharing specific storage areas across languages. The following sections discuss these features and pragmas.

For detailed pragma syntax and usage rules, see the *DEC Ada Language Reference Manual*. For more information on DEC Ada data representation, see Chapter 1 of this manual. See Chapter 9 of this manual for an example of sharing memory between processors.

4.5.1 Data Layout and Alignment in Mixed-Language Programs

Data layout and alignment is important for all objects and types that are shared in mixed-language programs, including objects and types involved in subprogram calls, as well as objects that are specifically designated as shared storage areas.

The default alignment of data on OpenVMS systems is as follows:

- On VAX systems, data is primarily byte aligned.
- On Alpha systems, data is primarily naturally aligned. In other words, 1-byte components are aligned on byte boundaries, 2-byte components are aligned on 2-byte boundaries, 4-byte components are aligned on 4-byte boundaries, and so on.

DEC Ada uses the default alignment for all types. To ensure that a particular alignment is always used for record and array types, use the DEC Ada pragma COMPONENT_ALIGNMENT. See Section 13.1a of the *DEC Ada Language Reference Manual* and Section 1.2.2 of this manual. To ensure that bit arrays or records of bits have a packed representation, specify them with a pragma PACK.

Similarly when passing strings to non-Ada routines, consider any differences that may exist between Ada string definitions and other-language string definitions. For example, C often expects string parameters to be null terminated. The DEC Ada predefined package C_TYPES provides support for handling null-terminated strings:

```
with C_TYPES; use C_TYPES;
with TEXT_IO; use TEXT_IO;
procedure PRINT_C_STRING is

   function RETURN_STRING return CHAR_POINTER;
   pragma INTERFACE(C, RETURN_STRING);
   pragma IMPORT_FUNCTION(RETURN_STRING, RESULT_MECHANISM => VALUE);

begin
   declare
      -- Declare string constants because the length of the
      -- returned string is not declared.
      --
      CHAR_PTR : constant CHAR_POINTER := RETURN_STRING;
      NULL_TERM_STR : constant NULL_TERMINATED.STRING :=
         NULL_TERMINATED.STRING'(NULL_TERMINATED.TO_STRING(CHAR_PTR));
      STR : constant STANDARD.STRING :=
         STANDARD.STRING'(NULL_TERMINATED.TO_STRING(NULL_TERM_STR));
   begin
      PUT_LINE(STR);
   end;
end PRINT_C_STRING;
------------------------------------------------------------

char *return_string ()
{
      return("Hello!");
}
```

For information on setting up Fortran common blocks, see Section 4.5.2. For examples of mixed-language programs involving shared data between Ada and C or between Ada and Fortran, see Sections Section 4.6 and Section 4.7.

4.5.2 Importing and Exporting Objects

DEC Ada provides the pragmas IMPORT_OBJECT and EXPORT_OBJECT to allow individual objects to be shared among mixed-language programs. These pragmas are equivalent to the GLOBAL and EXTERNAL attributes in Pascal, the GLOBALDEF and GLOBALREF attributes in PL/I, and global and **extern** declarations in C. You can also use the DEC Ada pragma INTERFACE_NAME to import objects from other-language programs. For example:

```
package IMPORTOBJ is
   C_INT: INTEGER;
   pragma IMPORT_OBJECT (C_INT);
end IMPORTOBJ;

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

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 C_INT in Ada is ");
   PUT(C_INT);
   NEW_LINE;
end IMPORTPROC;

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

int globaldef c_int = 10;
```

This example declares an object in Ada and a variable in C and uses the Ada pragma IMPORT_OBJECT to associate the name `c_int` with a global symbol. The C declaration assigns the value 10 to the location referenced by the global symbol, and the Ada main procedure prints it out.

The syntax and rules for using these pragmas are described in detail in Chapter 13 of the *DEC Ada Language Reference Manual*.

4.5.3 Sharing Common Storage Areas for Objects

For sharing common storage areas with other languages, DEC Ada provides the pragma COMMON_OBJECT. For example, you can use this pragma to associate Ada storage 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** declaration in C programs.

The syntax and rules for using the DEC Ada pragma COMMON_OBJECT are described in detail in Chapter 13 of the *DEC Ada Language Reference Manual*.

Unlike the programming languages that allow you to store several variables in a particular common block, DEC Ada allows only one object to be allocated in a particular storage area. For example, if you want to share one storage area with several Fortran common variables, you must declare an Ada record variable in which each component of the record corresponds to one Fortran variable:

```
C      Fortran declarations:

       INTEGER DAY, MONTH, YEAR
       CHARACTER*20 NAME
       COMMON /BDATE/DAY, MONTH, YEAR / /NAME
       END

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

-- Corresponding DEC Ada declarations:

type DATE is record
     DAY, MONTH, YEAR : INTEGER;
end record;
subtype NAME is STRING(1 .. 20);

BDATE : DATE;
ACCTNAME : NAME;

pragma COMMON_OBJECT (BDATE);
pragma COMMON_OBJECT (ACCTNAME, "$BLANK");
```

This example shows storage allocation in two different common storage areas:

- A common area that contains three integer components. This area is named BDATE in both the DEC Fortran and DEC Ada declarations.
- A common area that contains a 20-character array or string. In the Fortran declaration, this area is coded as a blank common block. In the equivalent Ada declaration, the area is coded as a string variable with the name ACCTNAME.

The Ada declarations use the pragma COMMON_OBJECT to establish the two common areas on the Ada side. The first pragma COMMON_OBJECT establishes the record variable BDATE as a common area with the external name (linker symbol) BDATE. The second pragma COMMON_OBJECT establishes the string variable ACCTNAME as a common area with the external name (linker symbol) "$BLANK". The name $BLANK must be quoted because it contains a dollar sign ($).

The layout of common data in other languages may depend on the system you are working on, compiler qualifiers, and so on. You must make sure that the alignments in the Ada code and other-language code match. For example:

- If the other language or system uses byte alignment, then you should specify a pragma COMPONENT_ALIGNMENT with the STORAGE_SIZE alignment choice to guarantee byte alignment. If you are working on a VAX system, the DEFAULT alignment choice also produces byte alignment.
- If the other language uses natural alignment, then you should specify a pragma COMPONENT_ALIGNMENT with the COMPONENT_SIZE alignment choice to guarantee natural alignment. If you are working on a Alpha system, the DEFAULT alignment choice also produces natural alignment.

  If you need to match Fortran common data where a size of 4 or fewer bytes is naturally aligned and a size greater than 4 bytes is aligned on the next 4-byte boundary, you must specify a pragma COMPONENT_ALIGNMENT with the COMPONENT_SIZE_4 option for the Ada types.

See Section 1.2.2 and the *DEC Ada Language Reference Manual* for more information about the pragma COMPONENT_ALIGNMENT. See Section 4.7 for more information about Fortran conventions and for an extended example that involves a Fortran common block. See the Fortran documentation for more information about Fortran common data alignment.

Also, common areas set up with the pragma COMMON_OBJECT correspond to OpenVMS program sections. Program sections established with the pragma COMMON_OBJECT have the following properties:

PIC, USR, OVR, REL, GBL, SHR, NOEXE, RD, WRT, NOVEC, ALIGN

On Alpha systems, program sections have the NOSHR property by default. They also have the NOMOD property.

Table 4–6 defines these properties. See the *OpenVMS Linker Utility Manual* for more information about program sections and their properties.

Table 4–6 Program Section Properties

Class	Meaning
ALIGN	Alignment
PIC/NOPIC	Position independent or position dependent
OVR/CON	Overlaid or concatenated
REL/ABS	Relocatable or absolute
GBL/LCL	Global or local scope
SHR/NOSHR	Shareable or nonshareable
EXE/NOEXE	Executable or nonexecutable
WRT/NOWRT	Writable or nonwritable
VEC/NOVEC	Vectors or no vectors (protection)
NOMOD/MOD[1]	Not modified or modified (initialized)
RD	Readability (reserved by Digital)
USR/LIB	User or library (reserved by Digital)

[1]On Alpha systems only.

4.6 Mixing C and Ada Code

When mixing Ada and C code, consider the following information:

- On VAX systems, C expects data to be byte aligned.

  On Alpha systems, C expects data to be naturally aligned. (These are the same default alignments as for Ada.)

- C passes all parameters by value. Array or string parameters are passed in a way that has the effect of passing them by reference. In C, you can pass parameters of any size by value. In Ada, parameters passed by the VALUE mechanism must have a compile-time size that is sufficiently small (32 bits on VAX systems; 64 bits on Alpha systems). On Alpha systems and in VAX C, parameters of the type LONG_FLOAT can also be passed by the VALUE mechanism.

- C returns all function results by value. You cannot return array results in C. C structs (equivalent to Ada records) that are returned by value are returned using the extra parameter method (like the extra parameter method used to return Ada function results with the REFERENCE mechanism; see Section 4.4.2).

- Returning a C pointer is not analogous to returning a value with the Ada REFERENCE result mechanism (which is done by the extra parameter method; see Section 4.4.2). When writing an Ada declaration for a routine that returns a pointer, you should declare a function that returns a result of an access type, and you should return the result by the VALUE result mechanism (the default). If the access type refers to a composite type, the type must be constrained.
- The C compiler generates uppercase names by default but lets you change the case to lowercase or mixed case with compiler qualifiers. The Ada compiler generates an uppercase external name for all imported or exported subprograms or objects. Lowercase or mixed-case names are incompatible with Ada names.

  You can work around the mixed-case C option either by spelling the names in the C program in uppercase, or by creating a jacket routine which has an uppercase name and which calls the mixed-case routine.

  See the *DEC Ada Language Reference Manual* for more information about external designators.
- C often converts float type parameters to double float unless you provide a function prototype. Use prototypes in C so that you can use the Ada type FLOAT or the DEC Ada types F_FLOAT or IEEE_SINGLE_FLOAT. If there is no prototype, you must use the Ada type LONG_FLOAT to accommodate for the C conversion. Be sure to use the same floating representation in both Ada and C (see Section 1.1.3 for more information about Ada floating representations).
- C often expects strings to be null terminated.
- C uses 0 for false and "not zero" for true. Ada uses 0 for false and 1 for true. If a C routine returns a boolean value, then the corresponding Ada declaration should return an integer value.
- In C routines, status flags are logically ORed together and passed as integers. In Ada, the analogous types are bit arrays or records of bits. You must specify a pragma PACK for the record or array type to ensure a contiguous layout of bits. Use an enumeration type to index the bits, and to give them the same names as the corresponding C names.
- Some C routines take unions as parameters. In Ada, declare one subprogram specification for each alternative of the union and make use of overloading.

- Many C Run-Time Library routines return data in static areas and are nonreentrant. You can use the DEC Ada package SYNCHRONIZE_NONREENTRANT_ACCESS with nonreentrant routines. See Section 7.4.9 for more information about handling nonreentrant routines.

Example 4–2 shows a situation where a global storage area is shared between an Ada procedure and a C routine. The storage area is represented in both Ada and C as a nested record structure. The C routine assigns values to the record components. The Ada procedure checks to be sure that the expected values were assigned.

Example 4–2 Sharing a Common Data Area with a C Program

```
with C_TYPES; use C_TYPES;
package GLOBAL_OBJ_PACKAGE is

   -- Declarations for the shared global data. The pragma
   -- COMPONENT_ALIGNMENT specifies natural alignment.

   type MAP is array (INTEGER range 1 .. 10) of SHORT_INT;

   type NESTED is
      record
         COMP    : INT;
         LIST    : MAP;
         CHARACT : CHAR;
      end record;
   pragma COMPONENT_ALIGNMENT (DEFAULT, NESTED);

   type REC is
      record
         FIRST  : NESTED;
         SECOND : INT;
      end record;
   pragma COMPONENT_ALIGNMENT (DEFAULT, REC);

   GLOBAL_OBJ : REC;
   pragma IMPORT_OBJECT (GLOBAL_OBJ);

   -- Declare the Ada interface to a C routine that will
   -- store known values in GLOBAL_OBJ when called.
   --
   procedure RESET_GLOBAL;
   pragma INTERFACE (C, RESET_GLOBAL);

end GLOBAL_OBJ_PACKAGE;

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

(continued on next page)

Example 4–2 (Cont.) Sharing a Common Data Area with a C Program

```
with GLOBAL_OBJ_PACKAGE; use GLOBAL_OBJ_PACKAGE;
with TEXT_IO; use TEXT_IO;
with C_TYPES; use C_TYPES;
procedure SET_GLOBAL_OBJ is

   FAILED_EXCEPTION : exception;

begin

   GLOBAL_OBJ.SECOND := -1;

   -- Test that the value was set correctly.
   --
   if GLOBAL_OBJ.SECOND /= -1 then
      raise FAILED_EXCEPTION;
   end if;

   -- Call the C routine that will store values in the global
   -- storage area.
   --
   RESET_GLOBAL;

   -- Check that the global storage area contains the values stored
   -- by RESET_GLOBAL.
   --
   if GLOBAL_OBJ.SECOND /= 1 then
      raise FAILED_EXCEPTION;
   end if;

   if GLOBAL_OBJ.FIRST.COMP /= 10 then
      raise FAILED_EXCEPTION;
   end if;

   for INDEX in GLOBAL_OBJ.FIRST.LIST'RANGE loop
      declare
         ELEMENT : SHORT_INT renames GLOBAL_OBJ.FIRST.LIST(INDEX);
      begin
         if INTEGER(ELEMENT) /= INDEX then
            raise FAILED_EXCEPTION;
         end if;
      end;
   end loop;
```

(continued on next page)

Example 4–2 (Cont.) Sharing a Common Data Area with a C Program

```
   if GLOBAL_OBJ.FIRST.CHARACT /= 'A' then
      raise FAILED_EXCEPTION;
   end if;

end SET_GLOBAL_OBJ;
------------------------------------------------------------------------

struct nested {
   int    comp;
   short  list[10];
   char   charact;
};

struct rec {
   struct  nested first;
   int            second;
} globaldef global_obj;

reset_global()
{
   int i;

   if (global_obj.second != 5)
      global_obj.second = 1;
   else
      global_obj.second = 2;

   global_obj.first.comp = 10;
   global_obj.first.charact = 'A';

   for (i = 0; i < 10; i++)
      global_obj.first.list[i] = i + 1;

}
```

Example 4–3 shows an Ada interface written to import a C routine that involves two array parameters. The array parameters are used, but not changed, during the execution of the routine.

The Ada interface is written expecting that the C routine does not change the values of its parameters. The interface specification uses only a pragma INTERFACE, indicating that the compiler should choose default mechanisms for passing the arrays and returning the result.

Example 4–3 Passing Arrays to C, Where the Array Values Are Not Changed

```
with C_TYPES; use C_TYPES;
with TEXT_IO; use TEXT_IO;
procedure CALL_INNER is

   package INT_TEXT_IO is new INTEGER_IO(INT);
   use INT_TEXT_IO;

   type INT_ARR is array(INTEGER range <>) of INT;
   A,B : INT_ARR(0 .. 9) := (1,2,3,4,5,6,7,8,9,10);
   function C_INNER(A,B : INT_ARR;
                    M   : INT) return INT;
   pragma INTERFACE(C, C_INNER);

begin

   for I in 0 .. 9 loop
      PUT(WIDTH => 3, ITEM => A(I));
   end loop;
   NEW_LINE;
   for I in 0 .. 9 loop
      PUT(WIDTH => 3, ITEM => B(I));
   end loop;
   NEW_LINE;
   PUT(C_INNER(A,B,10));
   NEW_LINE;
end CALL_INNER;
------------------------------------------------------------------------------

int c_inner(int a[], int b[], int n)
   {
   int i, c = 0;
   for (i=0; i<n; i++)
      c = c + a[i] * b[i];
   return c;
   }
```

Example 4–4 shows an Ada interface written to import a C routine that has an array parameter that is changed during the execution of the routine. In this case, the array is an **in out** parameter in the Ada subprogram specification: the parameter is marked as a parameter whose value will change.

The array parameter is explicitly passed by the REFERENCE mechanism to guarantee that the C routine receives the address of the array (which is what C expects). You must specify a mechanism for an **in out** parameter when the language is C.

Example 4–4 Passing an Array to C, Where the Array Value Is Changed

```
with C_TYPES; use C_TYPES;
with TEXT_IO; use TEXT_IO;
procedure CALL_NEG_ARRAY is

   package INT_TEXT_IO is new INTEGER_IO(INT);
   use INT_TEXT_IO;

   type INT_ARR is array (INTEGER range <>) of INT;
   A : INT_ARR(0 .. 9) := (1,2,3,4,5,6,7,8,9,10);

   procedure C_NEG_ARRAY (A : in out INT_ARR; M : INT);
   pragma INTERFACE (C, C_NEG_ARRAY);
   pragma IMPORT_PROCEDURE (INTERNAL => C_NEG_ARRAY,
                            MECHANISM => (REFERENCE, VALUE));

begin
   for I in 0 .. 9 loop
      PUT(WIDTH => 4, ITEM => A(I));
   end loop;
   NEW_LINE;
   C_NEG_ARRAY(A,10);
   for I in 0 .. 9 loop
      PUT(WIDTH => 4, ITEM => A(I));
   end loop;
   NEW_LINE;

end CALL_NEG_ARRAY;

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

void c_neg_array ( int a[], int n)
  {
  int i;
  for ( i = 0; i < n; i++)
    a[i] = -a[i];
  }
```

Example 4–5 shows an Ada interface for a C routine that passes and returns floating-point values (values of the type FLOAT on the Ada side). To ensure that the C compiler does not convert its type float to double float, a function prototype is defined for the C routine.

Example 4–5 Passing Floating-Point Values to C

```
with C_TYPES;
with TEXT_IO; use TEXT_IO;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure CALL_FLOAT is

   function C_FLOAT (X,Y : C_TYPES.FLOAT) return C_TYPES.FLOAT;
   pragma INTERFACE (C, C_FLOAT);
   pragma IMPORT_FUNCTION (INTERNAL => C_FLOAT,
                           EXTERNAL => FLOAT_EQ);

   A,B,X : C_TYPES.FLOAT;

begin

   A := 0.00000086;
   B := 0.00000092;
   X := C_FLOAT(A,B);
   PUT(X);
   NEW_LINE;

end CALL_FLOAT;

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

float float_eq(float x, float y)
   {
   return (x - y);
   }
```

4.7 Mixing Fortran and Ada Code

When mixing Ada and Fortran, consider the following information:

- The layout of data in Fortran depends on the compiler and compiler qualifiers you use. See the DEC Fortran documentation for specific information.

  You can use the DEC Ada pragma COMPONENT_ALIGNMENT to ensure the correct data layout for records and arrays. For example:

  – The COMPONENT_SIZE alignment choice ensures natural alignment (the default for all Ada records and arrays).

  – The STORAGE_SIZE alignment choice ensures byte alignment.

  – The DEFAULT alignment choice ensures the default alignment on whatever system you are working on (byte alignment on VAX systems, natural alignment on Alpha systems).

– The COMPONENT_SIZE_4 alignment choice ensures the alignment produced by the Fortran /ALIGN=COMMONS=STANDARD qualifier.

See the *DEC Ada Language Reference Manual* or Chapter 1 of this manual for information on the pragma COMPONENT_ALIGNMENT.

- Fortran passes most parameters by reference. String parameters are passed by descriptor.
- Fortran returns most results by value. Strings are returned by descriptor. You cannot return array or record results.
- On Alpha systems, Fortran complex data is returned by a mechanism that is not available in DEC Ada. There is no analogous restriction on passing complex data.

Example 4–6 shows an example that involves a Fortran common block. Note the following points about the example:

- The record types NESTED and STRUCT in the Ada package COMMON_PACKAGE match the record structures NESTED and STRUCT in the Fortran subroutine. The pragma COMPONENT_ALIGNMENT uses the STORAGE_UNIT alignment choice to cause byte alignment of the record components (the example assumes that the Fortran data is byte aligned).
- Because the pragma COMMON_OBJECT is used, the Ada program can initialize the variable OBJ, or it can leave initialization to the Fortran routine. In this case, one of the fields of the common variable OBJ is initialized by the Ada subprogram. When the Fortran routine is called, it initializes the entire object. When the Fortran routine returns, the Ada program verifies that the field it had initialized (OBJ.SECOND) has changed.

Example 4–6 Sharing a Fortran Common Block

```
package COMMON_PACKAGE is

   -- Declarations for the shared common block. Use the pragma
   -- COMPONENT_ALIGNMENT to set up byte alignment.

   type MAP is array (INTEGER range 1 .. 10) of SHORT_INTEGER;

   type NESTED is
      record
          COMP : INTEGER;
          LIST : MAP;
          CHAR : CHARACTER;
      end record;
   pragma COMPONENT_ALIGNMENT (DEFAULT, NESTED);

   type STRUCT is
      record
         FIRST  : NESTED;
         SECOND : INTEGER;
      end record;
   pragma COMPONENT_ALIGNMENT (DEFAULT, STRUCT);

   -- Declare the storage area for the common block.
   --
   OBJ : STRUCT;
   pragma COMMON_OBJECT (INTERNAL => OBJ,
                         EXTERNAL => "COMM");

   -- Declare the interface for the Fortran routine. The
   -- routine will store known values in OBJ (COMM on the
   -- Fortran side) when it is called.
   --
   -- Note that an identifier is specified for the external
   -- designator.
   --
   procedure RESET_COMM;
   pragma INTERFACE (FORTRAN, RESET_COMM);
   pragma IMPORT_PROCEDURE (INTERNAL => RESET_COMM,
                            EXTERNAL => FOR_ROUTINE);
end COMMON_PACKAGE;

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

with COMMON_PACKAGE; use COMMON_PACKAGE;
with TEXT_IO; use TEXT_IO;
procedure COMMON_OBJ_EXAMPLE is

   FAILED_EXCEPTION : exception;
```

(continued on next page)

Example 4–6 (Cont.) Sharing a Fortran Common Block

```
begin

   OBJ.SECOND := 5;

   -- Call the Fortran procedure that will store values in
   -- the common area.
   --
   RESET_COMM;

   -- Check that the common area contains the values stored
   -- by RESET_COMM.
   --
   if OBJ.SECOND /= 1 then
      raise FAILED_EXCEPTION;
   end if;

   if OBJ.FIRST.COMP /= 10 then
      raise FAILED_EXCEPTION;
   end if;

   for INDEX in OBJ.FIRST.LIST'RANGE loop
      declare
         ELEMENT : SHORT_INTEGER renames OBJ.FIRST.LIST(INDEX);
      begin
         if INTEGER(ELEMENT) /= INDEX then
            raise FAILED_EXCEPTION;
         end if;
      end;
   end loop;

   if OBJ.FIRST.CHAR /= 'A' then
      raise FAILED_EXCEPTION;
   end if;
end COMMON_OBJ_EXAMPLE;

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

C
C       This routine assigns values into the shared common area.  It
C       will be linked with the Ada code.  The Ada code calls it
C       and then checks the values in the common area.
C
C       Note that you need to make sure that the Fortran
C       and Ada data alignments match.

        SUBROUTINE     FOR_ROUTINE
```

(continued on next page)

Example 4–6 (Cont.) Sharing a Fortran Common Block

```
STRUCTURE      /NESTED/
   INTEGER     COMP
   INTEGER*2   LIST (10)
   CHARACTER   CHAR
END STRUCTURE

STRUCTURE /STRUCT/
   RECORD /NESTED/   FIRST
   INTEGER           SECOND
END STRUCTURE

RECORD /STRUCT/    OBJ

COMMON /COMM/      OBJ

INTEGER            I

IF (OBJ.SECOND .EQ. 5) THEN
   OBJ.SECOND = 1
ELSE
   OBJ.SECOND = 2
END IF

OBJ.FIRST.COMP = 10

OBJ.FIRST.CHAR = 'A'

DO I = 1,10
    OBJ.FIRST.LIST(I) = I
END DO

RETURN
END
```

Example 4–7 shows how to return complex numbers from a Fortran program on an Alpha system, using a jacket routine written in Fortran.

Example 4–7 Returning Complex Numbers from Fortran Programs on Alpha Systems

```
with TEXT_IO; use TEXT_IO;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure COMPLEX_EXAMPLE is

   type COMPLEX is record
      REAL_PART : FLOAT;
      IMAGINARY_PART : FLOAT;
   end record;

   X : COMPLEX;

   procedure GET_COMPLEX(X : out COMPLEX);
   pragma INTERFACE(FORTRAN, GET_COMPLEX);
   pragma IMPORT_PROCEDURE(GET_COMPLEX, MECHANISM => (REFERENCE));

begin

   GET_COMPLEX(X);

   PUT_LINE("The real part is: ");
   PUT(X.REAL_PART);
   NEW_LINE;
   PUT_LINE("The imaginary part is: ");
   PUT(X.IMAGINARY_PART);
   NEW_LINE;

end COMPLEX_EXAMPLE;
```

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

      SUBROUTINE GET_COMPLEX(DATA)

      COMPLEX*8   DATA

      DATA = (1.0, 0.1)

      RETURN
      END
```

5

Calling System or Other Callable Routines

DEC Ada provides a variety of features for calling OpenVMS system service, RMS, Run-Time Library, utility, and other callable routines from an Ada program:

- The package STARLET provides DEC Ada types, DEC Ada named numbers representing OpenVMS symbol definitions and DEC Ada operations for calling OpenVMS system service and RMS routines. The specification of this package is in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The package TASKING_SERVICES provides interface routines for calling OpenVMS system services that involve asynchronous system trap (AST) parameters. The specification of this package is in the DEC Ada library of predefined units (ADA$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 *DEC Ada Language Reference Manual* and is given in full in Appendix F of that manual. The specification of this package is also in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The generic package MATH_LIB (and the predefined DEC Ada MATH_LIB instantiations) provides routine interfaces for calling many of the OpenVMS Run-Time Library mathematics routines. The specifications for this package and these instantiations are in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The packages DTK, LIB, MTH, OTS, PPL, SMG, and STR provide DEC Ada types, DEC Ada named numbers representing OpenVMS symbol definitions, and DEC Ada operations for calling OpenVMS Run-Time Library routines. The specifications of these packages are in the DEC Ada library of predefined units (ADA$PREDEFINED).

- The packages CLI, NCS, LBR, and SOR provide DEC Ada types, DEC Ada named numbers representing OpenVMS symbol definitions, and DEC Ada operations for calling OpenVMS Command Language Interpreter, National Character Set, Librarian, and Sort/Merge Utility routines. The specifications of these packages are in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The package CONDITION_HANDLING provides a DEC Ada type for OpenVMS condition values, a set of functions for interpreting condition value components, and a set of interface routines for calling the OpenVMS Run-Time Library routines LIB$MATCH_COND, LIB$STOP, and LIB$SIGNAL. The specification of this package is in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The package SYSTEM_RUNTIME_TUNING allows you to tune aspects of run-time behavior that are normally controlled by the DEC Ada run-time library. The specification of this package is in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The package C_TYPES provides Ada equivalents for atomic C types, as well as types and subprograms for handling null-terminated strings. This package is designed to make C-related code easier to write (including calls to C Run-Time Library routines, and so on). The specification of this package is in the DEC Ada library of predefined units (ADA$PREDEFINED).
- The DEC Ada import pragmas allow you to write your own interfaces to callable routines. The DEC 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 4, and in Chapter 13 of the *DEC 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 OpenVMS Systems* for more information. For example:

```
$ ACS EXTRACT SOURCE STARLET
```

The following sections explain how to call OpenVMS system services, Run-Time Library, utility, and other callable routines, and give examples showing the use of the DEC Ada features for accomplishing such calls. You should be familiar with DEC Ada parameter passing and the OpenVMS calling standard, as well as with the DEC Ada import and export pragmas. See Chapter 4 of this manual and Chapter 13 of the *DEC Ada Language Reference Manual* for information on these topics.

For specific information on the calling standard and OpenVMS routines, see the appropriate OpenVMS documentation. For example:

- The *OpenVMS Calling Standard* presents the OpenVMS calling standard.
- The *OpenVMS Programming Interfaces: Calling a System Routine* gives general information about VMS system routines and explains how to call them.
- The *OpenVMS System Services Reference Manual* provides information on the OpenVMS system service routines.
- The *OpenVMS Record Management Services Reference Manual* provides information on the OpenVMS RMS routines.
- Individual run-time library manuals provide information on the OpenVMS Run-Time Library routines.
- The *OpenVMS Utility Routines Manual* provides information on the OpenVMS utility routines.

For specific information on callable interfaces for the various OpenVMS layered products, see the documentation for each product.

5.1 Using the DEC Ada OpenVMS System-Routine Packages

The DEC Ada predefined system-routine packages let you call system routines directly without having to specify your own interfaces. The following sections discuss the characteristics and use of these packages.

5.1.1 Parameter Types

The OpenVMS environment provides a set of data structures (OpenVMS usages) for denoting the OpenVMS data types used in OpenVMS system, OpenVMS Run-Time Library, and utility routines. Table 5–1 lists these data structures and gives their DEC Ada equivalents. For information on the underlying type representations, see Chapter 4. For information on the representation of the DEC Ada data types, see Chapter 1.

Note

Many of the equivalents are defined in the packages STARLET and CONDITION_HANDLING. For convenience, the OpenVMS Run-Time Library and utility packages define subtype equivalents for the STARLET and CONDITION_HANDLING types used in those packages.

Table 5–1 OpenVMS Data Structures

OpenVMS Data Structure	DEC 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

(continued on next page)

Table 5–1 (Cont.) OpenVMS Data Structures

OpenVMS Data Structure	DEC Ada Equivalent
floating_point	STANDARD.FLOAT STANDARD.LONG_FLOAT STANDARD.LONG_LONG_FLOAT[1] SYSTEM.F_FLOAT SYSTEM.D_FLOAT SYSTEM.G_FLOAT SYSTEM.H_FLOAT[1] SYSTEM.IEEE_SINGLE_FLOAT[2] SYSTEM.IEEE_DOUBLE_FLOAT[2]
function_code	STARLET.FUNCTION_CODE_TYPE
identifier	SYSTEM.UNSIGNED_LONGWORD
invo_context_blk	User-defined record
invo_handle	SYSTEM.UNSIGNED_LONGWORD
io_status_block	STARLET.IOSB_TYPE
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
mechanism_args	STARLET.CHFDEF2_TYPE
null_arg	SYSTEM.UNSIGNED_LONGWORD

[1]On VAX systems only.

[2]On Alpha systems only.

(continued on next page)

Table 5–1 (Cont.) OpenVMS Data Structures

OpenVMS Data Structure	DEC Ada Equivalent
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
time_name	STARLET.TIME_NAME_TYPE
transaction_id	**array**(1..4) **of** SYSTEM.UNSIGNED_LONGWORD
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	STANDARD.SHORT_INTEGER
word_unsigned	SYSTEM.UNSIGNED_WORD

5.1.2 Parameter-Passing Mechanisms

The OpenVMS system service, RMS, Run-Time Library, and utility routines conform to the OpenVMS calling standard. The DEC 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 OpenVMS documentation for detailed information on the passing mechanisms for parameters of system routines. Table 5–1 lists the DEC Ada equivalents for the OpenVMS data structures. See Chapter 4 for information on passing Ada parameters in mixed-language programs.

Note

Any parameter described in the OpenVMS documentation as a routine passed by reference is declared in the DEC Ada packages as a parameter of type ADDRESS that is passed by the VALUE mechanism. To pass the address of an Ada subprogram, you must first export the subprogram with one of the DEC Ada export pragmas (see Chapter 4 of this manual and Chapter 13 of the *DEC 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.

5.1.3 Naming Conventions

The following conventions are used in the DEC 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 ($) because dollar signs are not legal in Ada identifiers. In the OpenVMS Run-Time Library and utility-routine packages, all symbols have had their package-specific prefix removed. For example, you access LIB$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 5.1.4 for information on the naming conventions used for record types and initialization constants.

5.1.4 Record Type Declarations

The predefined system-routine packages contain type declarations for OpenVMS control blocks, masks, and so on. For example, the package STARLET declares the following control blocks used by OpenVMS RMS routines:

- The file access block (FAB)
- The record access block (RAB)
- The extended attribute block (XAB)
- The name block (NAM)

Many OpenVMS 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 (OpenVMS RMS file access block); see the *OpenVMS Record Management Services Reference 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,
                         . . .  ),
            FILLER_3 => FALSE,
            ESC      => FALSE,
            TEF      => FALSE,
            OFP      => FALSE,
            KFO      => FALSE,
            FILLER_4 => 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);
```

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 OpenVMS 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 OpenVMS 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 OpenVMS release, the name of the FILLER_4 component would change to FILLER_3, and FILLER_4 would no longer exist. So, 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 5–3 shows the use of some of the OpenVMS RMS control blocks declared in the package STARLET. The example is a program that maps a file to the first available space using the OpenVMS system service SYS$CRMPSC (Create and Map Section) and the OpenVMS RMS routine SYS$OPEN.

5.1.5 Default and Optional Parameters

As discussed in Chapter 4, the OpenVMS languages, and OpenVMS system service, RMS, Run-Time Library, and utility routines conform to a set of parameter-passing conventions called the OpenVMS calling standard. In accordance with the standard, each time an Ada subprogram or non-Ada routine is called, an argument list is passed. This list contains a count of the number of arguments, as well as the individual arguments themselves.

Many OpenVMS 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 argument count is 1, which indicates one argument. The single parameter is either the value zero, which indicates that the parameter is omitted, or it is 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.

The OpenVMS optional-parameter and the Ada default-parameter notions are not equivalent. The OpenVMS 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, using a zero to omit an argument can have a different interpretation from a zero passed by reference or a null string passed by descriptor.

Also, OpenVMS 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. OpenVMS RMS, Run-Time Library, and utility routines generally allow optional parameters to be indicated by shortened argument lists.

So, the following rules are true for the routines in *all* of the DEC 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 OpenVMS system service routines are declared with a default expression using the DEC 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 OpenVMS 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, 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 OpenVMS RMS routines in the package STARLET, as well as the OpenVMS 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. 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 *DEC Ada Language Reference Manual* for more detailed information on the rules for using this mechanism.

In summary, when calling a OpenVMS system service, RMS, Run-Time Library, or utility routine with optional parameters, you should follow these steps:

1. Consult the appropriate OpenVMS 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 DEC Ada package for the first routine interface declaration (if it is overloaded) to determine the 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. 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). So, 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 the following example. 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. So, 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 the following. 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 OpenVMS 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);
```

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 is 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 is passed in the argument list for ERR and the argument list is not truncated.

5.1.6 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 *OpenVMS System Services Reference Manual* describes these system services in more detail.

You can call asynchronous system services from a DEC Ada program by using tasks and the DEC Ada predefined pragma AST_ENTRY and AST_ENTRY attribute. See the *DEC Ada Language Reference Manual* and Chapter 7 for information on tasks and the pragma AST_ENTRY and AST_ENTRY attribute. Chapter 7 also gives several examples of programs where ASTs are handled.

Chapter 7 describes the package TASKING_SERVICES, which provides interface routines for calling services that involve AST parameters (SYSQIO, SYSGETJPI, and so on) from tasks. The subprogam specifications in the package TASKING_SERVICES have Ada bodies, and the NULL_PARAMETER attribute could not be used for optional parameters (see Sections Section 5.1.5 and Section 5.2.6). 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 let you increase the size of the AST packet pool. See Chapter 7 for more information on the AST packet pool and its limitations. See the specification of the package SYSTEM_RUNTIME_TUNING in the DEC Ada predefined library for more information about the package.

5.1.7 Calling Mathematical Routines

DEC Ada provides two packages of operations for calling OpenVMS Run-Time Library mathematical routines:

- The package MATH_LIB—provides interfaces for many of the OpenVMS 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 OpenVMS Run-Time Library format.

- The package MTH—also provides interfaces for many of the OpenVMS Run-Time Library mathematical routines and declares exceptions that can be raised. The interfaces match the OpenVMS 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, DEC Ada also provides instantiated versions of this package for the types FLOAT, LONG_FLOAT, and LONG_LONG_FLOAT (the type LONG_LONG_FLOAT is available on VAX systems only). See Appendix A for more information on these predefined instantiations.

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;
```

5.2 Writing Your Own Routine Interfaces

When you need to write your own interface to a callable routine from DEC 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

See the description of the routine in the appropriate OpenVMS or layered product documentation for more 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 DEC Ada import pragmas to import the routine so that you can call it as an Ada subprogram.

For example, the OpenVMS 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 DEC Ada.
- The data types (OpenVMS usages) required are as follows:

attr	mask_longword
tabnam	logical_name
lognam	logical_name
acmode	access_mode
itmlst	item_list_3

  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 as follows:

attr	Reference
tabnam	Descriptor
lognam	Descriptor
acmode	Reference
itmlst	Reference

- None of the parameters are themselves routines or addresses of routines.
- The *attr*, *acmode*, and *itmlst* parameters are optional parameters. SYS$TRNLNM is a OpenVMS system service, and system services require a fixed number of arguments. So, 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 DEC 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 DEC Ada interfaces for callable routine interfaces. For more information on the import pragmas and parameter-passing mechanisms, see Chapter 4. For complete examples of interfaces to system routines coded in Ada, see Section 5.5.

5.2.1 Parameter Types

If you are writing your own interface for a OpenVMS routine, see Table 5–1 for a list of the OpenVMS data structures and their DEC Ada equivalents. If you are writing your own interface for another kind of callable routine, see Chapter 4 for information on the DEC Ada equivalents for the OpenVMS data types defined in the OpenVMS calling standard. For information on the representation of the DEC Ada data types, see Chapter 1.

5.2.2 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. DEC Ada provides the pragma INTERFACE and the import pragmas IMPORT_PROCEDURE, IMPORT_FUNCTION, and INTERFACE_NAME to let you 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 Ada subprogram (see Section 5.2.5). DEC Ada provides the export pragmas EXPORT_PROCEDURE and EXPORT_FUNCTION to let you export Ada subprograms as procedures and functions respectively.

Because many system and utility routines return results *and* update their parameters, DEC 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) lets you write a DEC Ada interface that imports 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 lets you write a DEC Ada interface that exports 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. So, 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 4.4). 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 4. 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 DEC Ada, or you must specify the appropriate mechanisms in export pragma for the Ada subprogram.

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 OpenVMS Systems* for more information on that qualifier and those commands.

When you are working with the pragma EXPORT_VALUED_PROCEDURE, 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 4 of this manual and Chapter 13 of the *DEC Ada Language Reference Manual* for more information on using the import and export pragmas.

5.2.3 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 5–2 lists the Ada equivalents for the three most common OpenVMS access methods.

Table 5–2 DEC Ada Equivalents for OpenVMS Access Methods

OpenVMS Access Method	DEC 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 DEC Ada equivalents.

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. That parameter must have the mode **out** or a OpenVMS access method of modify. It usually corresponds to a condition value or equivalent returned by the applicable routine or subprogram.

5.2.4 Passing Parameters

Most callable routines (system or layered product) conform to the OpenVMS calling 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 4 for more information.

5.2.5 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 4 and Section 5.2.2). 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 imported or exported, the compiler issues a warning message.

Example 5–4 has an exported routine that is passed as a parameter to a OpenVMS Run-Time Library routine.

5.2.6 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 DEC Ada NULL_PARAMETER attribute, which places 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 DEC Ada predefined constant SYSTEM.ADDRESS_ZERO to place a zero value in the argument list. See the *DEC Ada Language Reference Manual* for more information about NULL_PARAMETER and ADDRESS_ZERO. See Section 5.1.5 for an explanation of how these mechanisms are used in the DEC 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 5.1.5 explains how overloading and FIRST_OPTIONAL_PARAMETER are used in the DEC Ada predefined system-routine packages. The *DEC Ada Language Reference Manual* gives detailed information on the FIRST_OPTIONAL_ PARAMETER mechanism. 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**.

5.3 Obtaining Symbol Definitions

Many of the global symbol definitions (condition values, and so on) you 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 *DEC 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 5.5.

```
with SYSTEM;
with CONDITION_HANDLING;
 . . .
procedure CREATE_LIB is

    . . .
   -- Initialize a constant with 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");
   RET_VAL    : CONDITION_HANDLING.COND_VALUE_TYPE;
    . . .

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;
```

5.4 Testing Return Condition Values

Many OpenVMS 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 DEC 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 OpenVMS Run-Time Library routine LIB$MATCH_COND, which are also provided in the package CONDITION_HANDLING.

For example, the following fragment from Example 5–1 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 DEC Ada predefined package CONDITION_HANDLING.

5.5 OpenVMS Routine Examples

Examples Example 5–1, Example 5–2, Example 5–3, Example 5–4, Example 5–5, Example 5–6, and Example 5–7 show the use of the package STARLET and import and export pragmas to make calls to various OpenVMS system service and Run-Time Library routines.

Example 5–1 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;

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);
```

(continued on next page)

Example 5–1 (Cont.) Calling SYS$TRNLNM Using the Package STARLET

```
   -- 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 5–2 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;
```

(continued on next page)

Example 5–2 (Cont.) Calling SYS$GETQUI Using the Package STARLET

```
   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),

      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),
```

(continued on next page)

Example 5–2 (Cont.) Calling SYS$GETQUI Using the Package STARLET

```
      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);


   -- 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;
```

(continued on next page)

Example 5–2 (Cont.) Calling SYS$GETQUI Using the Package STARLET

```
         -- 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 5–3 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;
```

(continued on next page)

Example 5–3 (Cont.) Calling SYS$CRMPSC Using the Package STARLET

```
   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);

   -- 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);
```

(continued on next page)

Example 5–3 (Cont.) Calling SYS$CRMPSC Using the Package STARLET

```
        -- 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 5–4 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

    -- 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;
```

(continued on next page)

Example 5–4 (Cont.) Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
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

    -- 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*.*";
```

(continued on next page)

Example 5–4 (Cont.) Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
    -- 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;

    -- 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;
```

(continued on next page)

Example 5–4 (Cont.) Calling LIB$FILE_SCAN and LIB$FILE_SCAN_END Using the Package LIB

```
    -- 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 5–5 Calling SMG Routines Using the Package SMG

```
-- This program demonstrates the use of the DEC 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);
```

(continued on next page)

Example 5–5 (Cont.) Calling SMG Routines Using the Package SMG

```
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);

   -- 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);
```

(continued on next page)

Example 5–5 (Cont.) Calling SMG Routines Using the Package SMG

```
-- 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);

-- 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);
```

(continued on next page)

Example 5–5 (Cont.) Calling SMG Routines Using the Package SMG

```
      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 5–6 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;
```

(continued on next page)

Example 5–6 (Cont.) Calling SYS$TRNLNM Using an Import Pragma

```
-- Define the OpenVMS 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;

-- 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 5–6 (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 5–6 (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 5–7 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 5–7 (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 5–7 (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;
```

6

Using CDD/Repository from DEC Ada

CDD/Repository lets you store data definitions so that they can be shared among various OpenVMS languages and OpenVMS data management products. As such, CDD/Repository provides the basis for a highly effective data management system.

CDD/Repository is an optional OpenVMS 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.*n* 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.*n* is called VAX CDD/Plus. It provides an additional set of features, including the ability to create distributed dictionary configurations. It uses a Common Dictionary Operator (CDO) format for internally representing data definitions, and provides the CDO utility for working with dictionaries and data definitions. VAX CDD/Plus is compatible and can be used with DMU dictionaries. VAX CDD/Plus also provides a call interface.
- Version 5.*n* or higher is called CDD/Repository. It includes all the features of VAX CDD/Plus Version 4.3 except for compatibility mode. Version 5.0 additionally provides the data integration capabilities required in software development environments.

The CDD/Repository documentation explains how to use CDD. In particular, *Using CDD/Repository on VMS Systems* provides tutorial information on building and maintaining repositories with the CDO utility.

DEC Ada provides a CDD translator utility to let you 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.

6.1 Using the DEC Ada-from-CDD Translator Utility

When you install DEC Ada, the files you need to use the DEC Ada-from-CDD translator utility are also installed. After DEC 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 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 OpenVMS 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 CDD/Repository path names (see *Using CDD/Repository on VMS Systems*). The different dictionary formats use different notation for the dictionary or repository 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 or repository definitions, a full path name begins with the dictionary anchor, which specifies the OpenVMS 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.
- CDD/Repository Version 5 supports both the backslash and period as separators. To use the backslash separator, define the logical name CDD$SEPARATOR to be "/". Then, you can use the backslash to separate descendant names and also refer to objects with periods in their names; for example, DISK:[JONES.CDD]MAIL_ORDER/SOURCE.C.

/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>
```

6.2 Equivalent DEC Ada and CDDL Data Types

The DEC Ada-from-CDD translator attempts to translate all CDD data types into equivalent DEC Ada data types. Some CDD data types are not native to DEC Ada. If a data definition contains an unsupported data type, the DEC 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 6–1 summarizes the mapping used by the translator between the CDD data types and the equivalent DEC Ada data types. Alpha includeS the same data types with some additions. For more information on the CDD data types, see the CDD/Repository documentation.

The specifications of the packages CDD_TYPES and SYSTEM are given in Appendix C. Alternatively, you can obtain the Ada source code for the package CDD_TYPES with the ACS EXTRACT SOURCE command. See *Developing Ada Programs on OpenVMS Systems* for more information on this command.

Table 6–1 Equivalent CDD and DEC Ada Data Types for OpenVMS Systems

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[1]	STANDARD.LONG_LONG_FLOAT
H_FLOATING COMPLEX	Unsupported type
UNSIGNED NUMERIC	Unsupported type

[1]On VAX systems only.

(continued on next page)

Table 6–1 (Cont.) Equivalent CDD and DEC Ada Data Types for OpenVMS Systems

CDDL Data Type	Ada Data 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
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

6.3 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 DEC 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:

```
define field order_num
    datatype is longword.
define field name
    datatype is text
    size is 20.
define field address
    datatype is text
    size is 20.
define field city
    datatype is text
    size is 19.
define field state
    datatype is text
    size is 2.
define field zip_code
    datatype is text
    size is 5.
define field item_num
    datatype is longword.
define field shipping
    datatype is f_floating.

define record info.
    order_num.
    name.
    address.
    city.
    state.
    zip_code.
    item_num.
    shipping.
end record.
```

To translate this definition to an Ada package, you first define the ADA$FROM_CDD command and then execute the command so that it extracts and translates the CDD record (assumed in this example to have the anchor directory DISK$:[USER.REPOS]. The definitions are stored in a repository directory called MAIL_ORDER). For example:

```
$ SET COMMAND SYS$LIBRARY:ADA$FROM_CDD.CLD
$ ADA$FROM_CDD/OUTPUT=INFO.ADA/PACKAGE DISK$:[USER.REPOS]MAIL_ORDER.INFO
```

You need to define the ADA$FROM_CDD command only once for any given terminal session. For your own convenience, you may want to define it in your LOGIN.COM file. See the *OpenVMS User's Manual* 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 DISK_USER_REPOS_MAIL_ORDER_INFO is

    -- CDD Path Name "DISK$:[USER.REPOS]MAIL_ORDER.INFO"

    type INFO_TYPE is
        record
            ORDER_NUM :       INTEGER;              -- signed 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 :        INTEGER;              -- signed longword
            SHIPPING :        FLOAT;                -- F_floating
        end record;

        for INFO_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 INFO_TYPE'SIZE use 624;

end;
```

You can then use this package in an Ada program as you would use any other Ada package. For example:

```
with DISK_USER_REPOS_MAIL_ORDER_INFO;
use DISK_USER_REPOS_MAIL_ORDER_INFO;
procedure USE_MAIL_DATABASE is
begin

   -- Work with the mail database using the type MAIL_ORDER_TYPE.

end USE_MAIL_DATABASE;
```

7
Tasking

Ada tasks are entities that execute in parallel. For example, you can use tasks:

- To take data concurrently from several sources
- To do terminal input-output and a series of calculations at the same time
- To call asynchronous OpenVMS system services

This chapter provides information on how to use DEC Ada tasks effectively, giving, in particular, information on how to use tasks in the OpenVMS environment.

If you are not familiar with Ada tasking, read Chapter 9 of the *DEC Ada Language Reference Manual* before reading this chapter. For information on the OpenVMS concepts presented in this chapter, see the *OpenVMS Programming Concepts Manual*. For information on DECthreads routines, see the *Guide to DECthreads*.

For information on the interaction of tasks with DEC Ada input-output facilities and exception handling, see Chapters Chapter 2 and Chapter 3.

7.1 Introduction to Using Ada Tasks on the OpenVMS Operating System

A *task* is an entity whose execution proceeds in parallel with the execution of other tasks. The Ada language lets you declare both task types and task objects.

An *environment task* is automatically created when you run a main DEC 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 *DEC 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 OpenVMS operating system.

Any task is said to depend on a number of masters. A *master* can be a block, task, or subprogram. For example if you declare a task object in a block, 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 either:

- The declaration of a task object, or
- The definition of the access type whose designated type is a task type

Control *cannot* leave a master until all of its dependent tasks have terminated. 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), DEC Ada automatically creates a *task control block* to manage the task. When the task is activated, DEC Ada creates a stack to be used by the statements that the task executes and lets the task compete for the processor on which your process is executing.

Because all tasks in any Ada program (including the main task) currently run in the context of a single process, control can switch from one task to another quickly. This switch can occur at or during any of the several machine instructions that make up an Ada program statement. The switch can occur midway through the execution of an Ada source line.

Because of task switching, you 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 7–1 reads in an array of integers and sorts them using a quick sort. The sorting is done by one task while another task (running in parallel) lets you 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). The *DEC Ada Language Reference Manual* fully defines these concepts.

VAX Systems Only

On VAX systems, you need to add the following statement to the beginning of the procedure TASKSORT:

```
pragma TIME_SLICE(0.3);
```

This statement enables round-robin task scheduling and gives each task an 0.3-second execution time slice. See Section 7.3 for more information about the pragma TIME_SLICE and its effect on DEC Ada tasks.

Example 7–1 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.

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

-- 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

   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 => ' ');
```

(continued on next page)

Example 7–1 (Cont.) Interactive Array Sort Using Tasks

```
-- Task to synchronize access to the array being sorted.
--
task GRANTOR is
   entry GRAB_ACCESS;
   entry RELEASE_ACCESS;
end GRANTOR;

-- 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;
```

(continued on next page)

Example 7–1 (Cont.) Interactive Array Sort Using Tasks

```
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;

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;
```

(continued on next page)

Example 7–1 (Cont.) Interactive Array Sort Using Tasks

```
        -- 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;

  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;
```

(continued on next page)

Example 7–1 (Cont.) Interactive Array Sort Using Tasks

```
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;

   -- 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;
```

(continued on next page)

Example 7–1 (Cont.) Interactive Array Sort Using Tasks

```
      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;
```

7.2 Task Storage Allocation

Each task created in your program requires storage. When your program creates a task, a task control block is allocated. When the task is activated, a stack is allocated. Because the OpenVMS 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.

7.2.1 Storage Created for a Task Object—The Task Control Block

When your program creates a task object, DEC Ada allocates a block of storage (a *task control block*) to keep track of that task's execution. For example, DEC Ada allocates a task control block 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;
```

DEC Ada deallocates the task control block when control leaves the immediate master (another task or a currently executing block or subprogram) on which the task depends and not when the task terminates. See Section 7.1 of this manual and Chapter 9 of the *DEC 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 following:

- The number of single entries in the task type
- The total number of *members* of all of its entry families
- The number of single entries that have been specified to receive ASTs (Asynchronous System Traps) (see Section 7.7)

In particular, if you specify an entry family with a 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 causes a large amount of storage to be allocated:

```
entry X (1..100_000);
```

You can estimate the number of pages to be allocated for a task control block with the following equation and constant definition table. You need to round up fractional results to a whole number:

- On VAX systems, the task-control-block size is calculated in terms of 512-byte pages, and you need to round up to the next page.
- On Alpha systems, the task-control-block size is calculated in terms of bytes.

$$TCB\_SIZE = \frac{FIXED\_AMOUNT + (E*C1) + (AST\_E*C2)}{D}$$

See Table 7–1 for a definition of terms.

Table 7–1 Definition of Terms in Task Control Block Size Equation

Constant	VAX Value	Alpha Value
FIXED_AMOUNT	3000 bytes	2000 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 (those entries declared with the pragma AST_ENTRY)	
C1	12.2	13.0
C2	28	40
D	512	1

For most task types (those having fewer than a few hundred total entries), the storage that the task control block consumes is relatively small. 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. You can also 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 (as shown in Example 7–2).

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.

As your program creates and terminates many tasks, storage for terminated tasks accumulates. 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.

Example 7–2 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.
```

This strategy 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.

7.2.2 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.

The task stacks are allocated as follows:

- The stack for a main task is allocated in the P1 region of the process in which the program is running, and it 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.

Note

On VAX systems, you can use the pragma MAIN_STORAGE to cause the stack for the main task to be allocated in the P0 region. See Section 7.2.2.2 for more information.

The task stack allocated for any DEC Ada task (including the main task) has two areas:

- A working storage area. This area is used during normal task execution for the storing of variables, call frames, and so on.
- A top guard area. This area is one or more pages at the top of the stack. This area is inaccessible to your program, and is designed to help you detect stack overflow.

On VAX systems, the task stack areas have the following characteristics:

- The default working-area size is 60 pages for tasks with fixed-size stacks.
- The top guard area is a set of 512-byte pages. The default size is 10 pages for tasks with fixed-size stacks.
- The default stack allocation for tasks with fixed-size stacks allows an additional 21 pages of stack for calls to non-Ada routines, which is adequate for most routines, including OpenVMS system service and Run-Time Library routines.
- Reading or writing the top guard area causes a hardware access violation (SS$_ACCVIO), which usually terminates your program immediately.
- If no top guard area exists, then you can accidentally overwrite the stack of another task in the following situations:
  - When a task with a fixed-size stack executes non-Ada code for which stack checking is not performed (see Section 7.2.3)
  - When storage size checks are suppressed when you compile the program (see Section 3.3)

To prevent this overwriting, use a nonzero top guard area so an access violation occurs when you run out of stack space (preventing you from overwriting the stack of another task).

On Alpha systems, the task stack areas have the following characteristics:

- The default working-storage area is 32K bytes for tasks with fixed-size stacks.
- The minimum (and default) size of the top guard area is one page.
- Reading or writing the top guard area causes a hardware access violation (SS$_ACCVIO). In most cases, this violation is treated as the predefined exception STORAGE_ERROR. See Chapter 3 for more information about exception handling.

Note

AST routines execute on the stack of whatever task is currently active. See Section 7.7 for more information on AST routines and tasks.

You may need to specify the sizes of a task's stack areas for any of the following reasons:

- A task is raising the exception STORAGE_ERROR, and you want to increase its working area.
- A task does not need all of its default stack allocation, and you want 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 suspect that some non-Ada routine might be overflowing the stack, and you want to increase the top guard area in an attempt to detect the overflow.
- On VAX systems, you have not called any non-Ada routines, and you are not having any stack overflow. (You have not suppressed checks and no task is raising STORAGE_ERROR.) You may want to decrease the top guard area and put the storage to other use.
- On VAX systems, in the case of a main task, you want to emulate the behavior of tasks on a VAX system running the VAXELN executive. See the *VAXELN Ada Programming Guide* for more information on VAXELN Ada.

You can use a number of mechanisms for controlling the size of a task's stack areas:

- Use the STORAGE_SIZE length representation clause attribute to control the size of the working storage area of any task stack except the main task stack.
- Use the DEC Ada pragma TASK_STORAGE to control the size of the guard area of any task stack except the main task stack.
- On VAX systems, use the pragma MAIN_STORAGE to control the storage allocated for a main task stack (and to force a fixed-size, P0 space stack allocation).

The following sections describe how to control task stack storage in more detail.

7.2.2.1 Controlling the Stack Sizes of Task Objects

To control the working storage area of the stack of a task object, you can apply the T'STORAGE_SIZE length 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*PAGE_SIZE;
```

Any task objects of this type have 300-page working storage areas.

If you specify a size of zero (bytes) with T'STORAGE_SIZE, a default stack size is used. Also, regardless of the size you specify, some additional space is allocated for task management purposes:

- On VAX systems, at least 21 pages are allocated.
- On Alpha systems, at least one page is allocated.

You can use the debugger to determine and tune the amount of storage you need for a stack working area (see *Developing Ada Programs on OpenVMS Systems*).

To control the top guard area of a task object, you can use the DEC 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 two pages:

```
pragma TASK_STORAGE(NEEDS_BIG_STACK, 2*PAGE_SIZE);
```

On VAX systems, you can set the top guard area to zero. For example:

```
pragma TASK_STORAGE(NEEDS_NO_GUARD, 0)
```

Any object of this type has a default working storage size (unless a representation clause was also specified) and no guard area.

The *DEC Ada Language Reference Manual* states 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.

If you anticipate using representation clauses or the pragma TASK_STORAGE later, you should use task types when you begin coding your tasking programs. 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 *DEC Ada Language Reference Manual* for more information on later declarations. See Chapter 13 of the *DEC Ada Language Reference Manual* for a description of the syntax and rules for using 'STORAGE_SIZE and the pragma TASK_STORAGE.

Example 7–3 shows the control of stack areas using 'STORAGE_SIZE and the pragma TASK_STORAGE.

Example 7–3 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 the minimum
   -- because the task NEEDS_BIG_STACK does not call outside Ada.
   -- On VAX systems, no guard pages are allocated; on Alpha
   -- systems, one guard page is allocated.
   --
   pragma TASK_STORAGE(NEEDS_BIG_STACK, 0);

   T : NEEDS_BIG_STACK;
```

(continued on next page)

Example 7–3 (Cont.) Controlling the Size of a Task's 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;
```

7.2.2.2 Controlling the Size of a Main Task Stack (VAX Systems Only)

Main task stacks usually have no definite limit and are automatically expanded as needed in the OpenVMS environment. However, DEC Ada provides the pragma MAIN_STORAGE, which 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). It is generally useful in the OpenVMS environment when you need to simulate the behavior of a VAXELN main task when you are working with DEC Ada and an OpenVMS target. See the *VAXELN Ada Programming Guide* for more information on VAXELN Ada.

The pragma MAIN_STORAGE has two parameters, WORKING_STORAGE and TOP_GUARD, which let you 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;
```

The working storage area of the main program in this example is limited to 100*512 bytes (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.

The debugger can help you to determine and tune the amount of storage you need for a stack working area. See *Developing Ada Programs on OpenVMS Systems* for more information.

See Chapter 13 of the *DEC Ada Language Reference Manual* for a description of the syntax and rules for using the pragma MAIN_STORAGE.

7.2.3 Stack Overflow and Non-Ada Code

DEC Ada raises the exception STORAGE_ERROR when an attempt is made to overflow either the main stack or an Ada task stack. The default stack storage allocated for each non-Ada call should also be adequate protection against stack overflow for most non-Ada routine calls (see Section 7.2.2).

When you call a non-Ada routine from an Ada program, the stack of the main task or an individual task may overflow. Many non-Ada routines do not check for this. The Ada program is not be able to detect overflow because the exception STORAGE_ERROR has not been raised. Such an undetected stack overflow may 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.

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 7.2.2 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 raise a hardware access violation SS$_ACCVIO exception or a STORAGE_ERROR exception (see Section 3.1.1), which usually terminates your image immediately.

The debugger can be helpful in detecting stack overflow. The debugger performs an automatic stack check for you and can display the amount of stack space in use in any task. The DEC Ada predefined package GET_TASK_INFO also provides operations that you can use to obtain information about the currently executing task. For further information, see *Developing Ada Programs on OpenVMS Systems*.

7.3 Task Switching and Scheduling

DEC Ada implements the Ada language requirement that when two tasks are eligible for execution and they have different priorities, the lower priority task does not execute while the higher task is waiting. The DEC Ada run-time library keeps a task running until either the task is suspended or a higher priority task becomes ready.

Note

This chapter uses the term "suspend" 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 OpenVMS system service $SUSPND.

Two scheduling strategies are available for DEC Ada tasks:

- *First-in-first-out (FIFO), with preemption* is the default strategy on VAX systems. With this strategy, 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.
- *Round-robin, with preemption* is the default strategy on Alpha systems. With this strategy, tasks of equal priority take turns at the processor. A task is run for a certain period of time, then placed at the rear of the ready queue for its priority level.

In both cases, the term *with preemption* means that DEC Ada preempts a running task if a higher priority task becomes ready. This behavior is required by Ada rules (see Section 9.8 of the *DEC Ada Language Reference Manual*). The preempted task is placed at the front of the ready queue for its priority level. When the higher priority task suspends, the preempted task resumes execution. The preemption of a lower priority task does not imply any cycling of the ready queue for that priority.

The FIFO scheduling strategy increases program repeatability and helps you debug your program. 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).

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 *never* get to run if the running task does not become suspended.

Round-robin scheduling prevents nonsuspending tasks from capturing the processor and simulates more realistically tasking on parallel processors. It also tends to make tasks of equal priority execute in an arbitrary order and stresses the tasking logic in your program.

There are at least two ways in which you can control task scheduling:

- You can use the pragma PRIORITY to give the more important tasks higher priorities and increase their responsiveness.
- You can use the pragma TIME_SLICE (available on all OpenVMS systems) or the procedure SYSTEM_RUNTIME_TUNING.SET_TIME_SLICE (supported on VAX systems only) to enable or disable round-robin scheduling.

The following sections discuss the pragma PRIORITY and time slicing in more detail.

7.3.1 Controlling Task Priorities

To let you control task priorities, the Ada language provides the pragma PRIORITY. For example, the following statements set the priority of the task IMPORTANT_TASK to 14:

```
task IMPORTANT_TASK is
   pragma PRIORITY(15);
end IMPORTANT_TASK;
```

The pragma PRIORITY can appear only in a task specification or in the outermost declarative part of a main subprogram. See Chapter 9 of the *DEC Ada Language Reference Manual* for a description of the syntax and rules for using this pragma.

On VAX systems, the range of possible DEC Ada task priorities is from 0 to 15. In the absence of this pragma, DEC Ada tasks have a default midrange priority of 7.

On Alpha systems, the range of possible task priorities is also from 0 to 15. However, if you do not explicitly specify a priority for a given task, the priority for that task is considered to be undefined. In DEC Ada, a task with an undefined priority competes fairly with other tasks, usually behaving as if it had a midrange priority (between 7 and 8).

A task whose activation occurs as part of the execution of another task (including the main task) inherits the priority of the parent task.

On all OpenVMS systems, task priority has no effect on the priority of your process. The process priority applies to the execution of every task in your program.

7.3.2 Using Time Slicing

The DEC Ada time-slicing features enable (or disable) round-robin scheduling (see Section 7.3). 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 stresses the tasking logic in your program.

You can control time slicing on OpenVMS systems with the following features:

- The pragma TIME_SLICE (available on all OpenVMS systems)
- The procedure SYSTEM_RUNTIME_TUNING.SET_TIME_SLICE (supported on VAX systems only)

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. The pragma TIME_SLICE has an effect only if it appears in the outermost declarative part of a main program. See Chapter 9 of the *DEC Ada Language Reference Manual* for a complete description.

See the specification of the package SYSTEM_RUNTIME_TUNING in Appendix C for information on using the SET_TIME_SLICE procedure.

Time-slicing is implemented as follows:

- On VAX systems, when you specify a positive, nonzero time slice with either the pragma TIME_SLICE or the procedure SYSTEM_RUNTIME_TUNING.SET_TIME_SLICE, you change the default FIFO scheduling strategy to round-robin scheduling. Tasks of the same priority take turns at the processor for the specified amount of time (in seconds).

  When you specify a negative or zero value, you disable time slicing (change the scheduling strategy back to FIFO). On VAX systems, both the pragma TIME_SLICE and the procedure SYSTEM_RUNTIME_TUNING.SET_TIME_SLICE let you control the execution time (the size of the time slice) for any particular task.

- On Alpha systems, round-robin scheduling is the default. You can change to FIFO scheduling by specifying a negative or zero value for the pragma TIME_SLICE. You can enable (guarantee) round-robin scheduling by specifying a positive value for the pragma TIME_SLICE. You cannot switch back and forth between FIFO and round-robin scheduling in the same program.

If you enable round-robin scheduling, you increase fairness while increasing task switching overhead. (On VAX systems, the overhead increases for smaller

time-slice values.) Debugging is also more difficult. On VAX systems, time-slice values below 0.01 second do not result in faster time slicing because the smallest time increment supported by the OpenVMS operating system is 0.01 second.

7.4 Special Tasking Considerations

Use of tasks in an Ada program requires some care. 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:

- Passive tasks
- Deadlock
- Busy waiting
- Tentative rendezvous
- Delay statements
- Abort statements
- Interrupting program execution with Ctrl/Y
- Shared variables
- Reentrancy

7.4.1 Passive Tasks

A *passive task* is a task that follows one or more of a set of requirements. The following requirements apply to the task specification:

- No entry families
- No AST entries
- Not allowed within a generic context
- No more than 62 entries

The following requirements apply to the declarative part of the task body:

- No stubs
- No access type declarations
- No generic instantiations
- No task object declarations

- No package bodies

The following conditions apply to the body of a task:

- No accept or select statements in its initial section or inside the body of an accept statement.
- Final statement must be an unconditional loop. Within the loop, the following applies:
  - Nested accept statements are not allowed.
  - Each entry must have exactly one accept statement within the task body.
  - No exception handlers for the task body are allowed. However, exception handlers within nested declarative parts are acceptable.
  - The unconditional loop must take one of the following forms:

    a. Unconditional loop containing simple accept statements with optional bodies and no other statements. For example:

```
loop
    accept e1(...) do  -- optional body
    ...
    end e1;

    accept e2(...) do  -- optional body
    ...
    end e2;
end loop;
```

    b. An unconditional loop containing a selective wait with optional guards, accept bodies, and an optional terminate alternative. No other statements are allowed. Also, no delay alternative is allowed. The guard expression must not contain user-defined subprogram calls.

       The following is an example of an unconditional loop:

```
loop
    select
        when <condition> =>   -- optional guard
            accept e1(...) do -- optional accept body
            ...
            end e1;
    or
        when <condition> =>   -- optional guard
            accept e2(...) do -- optional accept body
            ...
            end e2;
    or
        when <condition> =>   -- optional guard
        terminate;            -- optional terminate
                              -- alternative
    end select;
end loop;
```

The guard expression can include calls to the functions in packages SYSTEM and STANDARD, and it can include calls to the predefined operators associated with types. User-defined operator calls are not allowed.

The only requirement on the code executing within an accept body is that it not contain any select statements.

Note

Since the accept body is executed in the context of the task making the entry call, it is executed on the stack of the task making the entry call. When making an entry call to a passive task, make sure that the stack of the calling task is large enough to handle the passive task's accept body.

In addition, a passive task must adhere to one of the following formats:

-

```
task body I_AM_PASSIVE_1 is
    <declarative part>
begin
    <initial section>
    loop
        <one select statement>
    end loop;
end;
```

•

```
task body I_AM_PASSIVE_2 is
        <declarative part>
begin
    <initial section>
    loop
        <one or more accept statements>
    end loop;
end;
```

7.4.1.1 Passive Tasks and Rendezvous

Taking advantage of DEC Ada support for passive tasks can significantly improve the performance of rendezvous in your programs. A task rendezvous (consisting of an entry call to a passive task) is accomplished with no context switching overhead. Instead, the accept body is executed in the context of the task making the entry call.

In an ordinary rendezvous, the caller must block and pass control of the processor to the task it is calling. The called task must execute the rendezvous, block, and return control to the caller task.

No blocking or passing of control occurs when making an entry call to a passive task. Control of the processor does not pass to the passive task. Therefore, rendezvous with passive tasks is significantly faster.

To take advantage of passive tasks, you need to compile the packages containing both the task specification and body using the /OPTIMIZE qualifier. The /OPTIMIZE qualifier indicates that all tasks that can be made passive and that are not declared with the pragma PASSIVE(NO) be made passive. If you specify the /OPTIMIZE qualifer, the compiler implements all tasks that meet the above requirements as passive tasks, regardless of whether or not their specifications contain a pragma PASSIVE. (See Section 7.4.1.2.

The /NOOPTIMIZE qualifier prevents the compiler from making any tasks passive (even those containing a pragma PASSIVE).

Not all tasks can be passive tasks. The best candidates are those tasks that act as servers or as protectors of shared resources. These tasks commonly consist of a select statement within an infinite loop. A typical server task accepts an entry call and then loops back to wait for the next entry call.

7.4.1.2 Pragma PASSIVE

You can use the pragma PASSIVE in the following ways:

- To tell the compiler that you would like a particular task to be passive
- To explicitly prevent the compiler from making a task passive

When you specify a pragma PASSIVE and the task does not meet the requirements for a passive task, the task is not made passive.

The form of pragma PASSIVE is as follows:

```
pragma PASSIVE [(passive_form)];
passive_form => SEMAPHORE | NO
```

You must specify the pragma PASSIVE within a task specification. The specification can be for a task type or for a single task. The containing task must conform to requirements listed in the preceding sections.

The following forms are equivalent and are considered assertions that the containing task is passive and that optimization of context switch with this task is permitted and desired:

```
pragma PASSIVE;
pragma PASSIVE(SEMAPHORE);
```

The following form is an assertion that rendezvous with the containing task should not be optimized:

```
pragma PASSIVE(NO);
```

Specify either a pragma PASSIVE (SEMAPHORE) or a pragma PASSIVE with no arguments in a task specification to cause the compiler to generate warning messages if the specified task cannot be made passive.

Specifying a pragma PASSIVE is useful in determining why the compiler cannot make a task passive. It is also useful if you are concerned that future changes to a task may prevent it from being passive. The compiler issues warning messages when changes to the task prevent it from being treated any longer as passive.

A pragma PASSIVE (NO) explicitly directs the compiler not to treat a task as passive, even if it meets all of the specified requirements.

Instead of specifying a pragma PASSIVE, you can use compilation notes to determine whether or not the compiler is treating a task as passive. To generate compilation notes, specify the /WARNINGS=COMPILATION_NOTES qualifier. During compilation, the compiler issues compilation messages stating which tasks are passive and which are not.

7.4.2 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. The debugger also provides special task debugging commands that can help you detect deadlocks (see *Developing Ada Programs on OpenVMS Systems*).

The following are some of the more common forms of Ada deadlock:

- *Exception-induced*. 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 7–4 shows an exception-induced deadlock.

- *Self-calling*. 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 7–5 shows self-calling deadlock.

Example 7–4 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 7–5 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;
```

(continued on next page)

Example 7–5 (Cont.) A Self-Calling Deadlock

```
   task body T is
   begin
      P(Y);                -- Never returns.
      accept E;
   end T;

begin
   null;
end SELF_CALL;
```

- *Circular-calling.* 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 7–6 shows circular-calling deadlock.

Example 7–6 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;
```

(continued on next page)

Example 7–6 (Cont.) A Circular-Calling Deadlock

```
begin
   null;
end CIRCULAR_CALL;
```

- *Dynamic-circular-calling.* Occurs when a series of entry calls forms a circle as in circular-calling or self-calling deadlocks. However, at least one of the calls is a timed or conditional entry call in a loop that completes only if the rendezvous occurs. With dynamic-circular-calling deadlock, at least one task is executing, but no progress can be made. Example 7–7 shows a dynamic-circular-calling deadlock.

Example 7–7 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;
```

7.4.3 Busy Waiting and Non-Ada Code

Sometimes called a "flag" or a "spin lock," *busy waiting* is a programming technique that repeatedly tests a variable to determine if some event has occurred. When the event does occur, another instruction sequence is presumed to execute and set the flag, ending the looping.

Busy waiting is sometimes desirable when an event occurs 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.

However, busy waiting has some undesirable characteristics:

- 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 OpenVMS operating system, this usually means that the process is using processor resources that another process could use to advantage.
- 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 in effect (see Section 7.3), deadlock develops because the busy-waiting task does not suspend and no other task (including the flag-setting task) can be scheduled. (This behavior is in accordance with Ada rules.) The situation can be improved slightly if time slicing is in effect. 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. DEC Ada does not use busy waiting, so if your program uses only DEC 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 by eliminating all specifications of the pragma PRIORITY) and by enabling time slicing (see Section 7.3).

7.4.4 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.

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. Therefore, tentative rendezvous constructs require special care.

7.4.5 Using Delay Statements

DEC Ada implements the delay statement as a call to the OpenVMS system service SYS$SETIMR. Each delay statement places an entry in the system timer queue, which, in turn, affects the OpenVMS operating system Timer Queue Entry Limit (TQELM) quota. Each delay statement also makes use of the SYS$SETIMR routine's ASTADR parameter, which specifies an AST routine. 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. The call stalls until an active delay expires, and the delay does 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 time slicing is enabled (see Section 7.3).

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 enables time slicing. 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 *OpenVMS System Manager's Manual* gives details on how to adjust these quotas.

7.4.6 Using Abort Statements

Be careful when you use abort statements. An abort statement can terminate a task when it should not be terminated and 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 OpenVMS operating system has written the result data. The result data could be written in some unexpected part of your program's data area.

DEC 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 *DEC Ada Language Reference Manual* for a list of these points.

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 executes to completion even though the calling task has been aborted. In this way, synchronous abort avoids problems that may result because non-Ada routines typically are not programmed to work correctly if they are 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 becomes 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. **Delay** 0.0 is a low-overhead means of ensuring that completion can occur.

7.4.7 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 OpenVMS 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 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.

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 immediately, the desired effect may not occur for awhile 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 or Alpha 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 do not have a chance to execute.

For example, DEC Ada provides an exit handler for the input-output packages. If you enter the STOP command to interrupt input-output, the DEC Ada handler does 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 DEC Ada predefined package CONTROL_C_INTERCEPTION, which lets you 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, CONTINUE, EXIT, or a DCL command.
ADA_Ctrl/C> DEBUG
DBG>
```

See Appendix C for the package specification of CONTROL_C_INTERCEPTION.

7.4.8 Using Shared Variables

The code that an Ada compiler generates may store the value of a variable in several, one, or no places in the memory of the machine (see Chapters Chapter 1 and Chapter 8). Unless instructed otherwise, the compiler believes that it can detect all attempts to read or write a variable and arranges to have each of those attempts access the correct places.

In the absence of a pragma SHARED or VOLATILE, the compiler makes assumptions based on the following rules:

- A variable that is read by a task is not written by another task until the reading task reaches a synchronization point.
- A variable that is written by a task is not 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 *DEC 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, you must specify the pragma SHARED or VOLATILE for that variable. Otherwise, your program is erroneous.

The Ada language defines the pragma SHARED. DEC Ada defines the pragma VOLATILE. Chapter 9 of the *DEC Ada Language Reference Manual* gives the Ada language assumptions about shared variables and gives the usage rules and syntax for the pragmas SHARED and VOLATILE.

In particular, these pragmas require that the named variable be declared by an object declaration. For the pragma SHARED, the variable must be of a scalar or access type. For the pragma VOLATILE, the variable can be of any type. For example:

```
pragma SHARED(INTEGER_OBJECT);
pragma VOLATILE(ANY_OBJECT);
```

The pragma SHARED 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. The pragma SHARED also 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, DEC Ada guarantees that every read or update of a shared variable is a synchronization point. DEC 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).
- On VAX systems, an *interlocked* instruction is executed.

  On Alpha systems, a *memory barrier* instruction is executed.

This instruction or sequence of instructions is executed so that all 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).

DEC 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. On Alpha systems, a memory barrier is executed before each read.

DEC Ada ensures the indivisibility of reads and updates of variables specified by the pragma SHARED as follows:

- On VAX systems, 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.

  On Alpha systems, all scalar or access variables are allowed.

- On VAX systems, by allocating longword-aligned longwords for all shared variables whose storage size is larger than a byte.

  On Alpha systems, by naturally aligning all shared variables.

- By using a restricted set of VAX or Alpha 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. This must happen 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 is 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 repeats forever if it is checking a value of PTR that was read before the loop was entered or if it is checking a value of PTR that was stored in a register or optimized away.

A pragma VOLATILE or SHARED ensures that the compiler stores the value of PTR in an actual memory location, ensuring that any new value written to that location is fetched when the value of PTR is checked.

If PTR is declared with the pragma VOLATILE, then the loop repeats 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 does 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, guaranteeing that the values for INFO and INFO_ VALID are both 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.

7.4.9 Reentrancy

Note

Within this section and subsections, the term reentrant denotes full reentrancy, unless explicitly qualified.

In most languages on OpenVMS systems, the following four kinds of reentrancy are possible:

- A routine is *serially reentrant* if it must execute to completion before it can be called again. FORTRAN routines are usually serially reentrant.
- A routine is *recursively reentrant* if it:

  1. Executes to the point of another call to itself
  2. Makes the call to itself (a recursive call)
  3. Continues to make recursive calls until a statement is executed or a condition occurs that ends the recursion

  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.
- A routine is *AST reentrant* if, at a random point during the execution of a routine, an AST can occur and the routine can be reentered (by the AST call). The OpenVMS operating system does not normally allow more than one AST service routine to be called at a time for any given access mode.

If a routine is AST-reentrant, it can 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.

Note

When calling non-Ada routines from DEC Ada tasks, the results are unpredictable if the routines are not fully reentrant. The following routines are reentrant:

- All OpenVMS system service and most OpenVMS 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.

The following routines are nonreentrant:

- Any routine that modifies variables outside its immediate scope or that modifies variables allocated in static storage is potentially nonreentrant.
- Any language-dependent run-time library routines may be nonreentrant. For example, the FORTRAN run-time library is only AST (not fully) reentrant. Also, most C run-time library routines are nonreentrant.
- X Windows and Motif routines are nonreentrant.

Reentrancy can be an issue for both Ada and non-Ada routines. For example, the subprogram in Example 7–8 shows that if you allow a nonreentrant Ada subprogram (or non-Ada routine) to be reentered, the results can be unpredictable.

Example 7–8 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 7–8, 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:

1. The subprogram NONREENTRANT is called by task A, which passes a value of 3 for I.
2. A is interrupted just before statement S2 because a higher priority task, B, has become ready.
3. 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.

The following Ada coding techniques avoid the problem shown in Example 7–8:

- Write the routine or subprogram so that it is reentrant.
- Serialize the calls to the nonreentrant routine or subprogram (see Example 7–10).

- Ensure that only one task can call the nonreentrant routine or subprogram.

The following sections discuss these techniques in more detail.

7.4.9.1 Coding Reentrant Ada Subprograms

To code a reentrant subprogram in DEC 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). Imported non-Ada routines, can be reentered if imported several times in the same Ada program or if imported once and then called from different tasks.

In Example 7–9, the function NONREENTRANT from Example 7–8 is rewritten so that it is reentrant. Advantage is 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 7–9 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;
```

7.4.9.2 Ensuring that Nonreentrant Routines are Called by One Task at a Time

You can ensure that nonreentrant routines are called by only one task at a time by using one of the following methods:

- Structuring your program appropriately. 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.
- Using the lock and unlock procedures in the DEC Ada package SYNCHRONIZE_NONREENTRANT_ACCESS.

If you use the first method, then your Ada code uses any potentially nonportable features.

The GLOBAL_LOCK and GLOBAL_UNLOCK procedures in the DEC Ada package SYNCHRONIZE_NONREENTRANT_ACCESS establish a lock, which you code explicitly, according to the locking protocol supported by the package. These procedures are especially useful in the situation where you know that some (but not all) calls to a routine are nonreentrant. For example:

```
with SYNCHRONIZE_NONREENTRANT_ACCESS;
use SYNCHRONIZE_NONREENTRANT_ACCESS;
 . . .
   GLOBAL_LOCK;
   <call to something nonreentrant>
   GLOBAL_UNLOCK;
```

The package specification explains how the procedures work in more detail. See Appendix C for more information on obtaining the package specification.

Note

Failure to follow the locking protocol may cause your program to produce unexpected results. For example, if two or more tasks in your program call a particular routine concurrently, then the data structures used by the routine may become corrupted. The result may be an access violation or another error in your program.

7.4.9.3 Serializing Calls to Nonreentrant Code

You can use a task to prevent reentry by serializing the calls to the nonreentrant code so that it cannot be reentered. This technique applies especially to existing nonreentrant Ada subprograms, non-Ada routines, or software over which you have no control. Example 7–10 shows one way to perform serialization.

Example 7–10 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;

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

(continued on next page)

Example 7–10 (Cont.) Using a Serializing Task to Prevent Reentry

```
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;

   function ADD_ONE (I : INTEGER) return INTEGER is
      RESULT : INTEGER;
   begin
      SERIALIZER.DO_CALL(I, RESULT);
      return RESULT;
   end;

end FIX_IT;
```

In Example 7–10, 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 are not attempted.

7.5 Calling OpenVMS System Service Routines from Tasks

DEC Ada provides the package STARLET (see Chapter 5) as well as import-export pragmas (see Chapter 4) to let you call OpenVMS system services and make OpenVMS RMS requests directly from an Ada program. DEC Ada also 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 7.7.

7.5.1 Effects of System Service Calls on Tasks

When you call OpenVMS 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 *OpenVMS Programming Concepts Manual*). To DEC Ada, a task that has entered an OpenVMS wait state appears to be continuing to execute (because DEC Ada does not intercept system services). DEC Ada does not know that the task is blocked.

The only tasks that can execute while the system service is executing are:

- Tasks that have higher priorities than the calling task
- If time slicing is in effect (see Section 7.3), 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 DEC 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 OpenVMS programs. Indeed, if the request is satisfied quickly, allowing any other task to execute may be wasted effort.

You may, however, wish to increase concurrency and let tasks of lower priority execute while a higher priority task is in a OpenVMS 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.

DEC Ada provides you with two methods for increasing concurrency during a OpenVMS system service wait interval:

- Have tasks that call time-consuming system services use asynchronous system services or asynchronous OpenVMS RMS services. Then your program can do other work until it has to handle the resulting OpenVMS ASTs that signify completion of the request. Handling ASTs is a general and powerful way to increase concurrency, but it also requires more detailed programming. See Section 7.7.
- Use the OpenVMS system-routine operations provided in the DEC 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 OpenVMS system service and RMS routines.

  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 package hides details of AST handling.

While this package can help you increase concurrency in many cases, 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 7.7.

The specification of the package TASKING_SERVICES is presented in Appendix C.

7.5.2 System Services Requiring Special Care

Certain system services are likely to interfere with Ada programs that use tasks, ASTs, or the package TASKING_SERVICES (see Section 7.5.1). 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 an OpenVMS process, these services have a global effect on all tasks of the program. For example, SYS$SETAST prevents delivery of ASTs. Because the DEC Ada tasking implementation relies heavily on the use of ASTs (they are used to implement task scheduling, input-output, and so on), disabling ASTs with SYS$SETAST can cause deadlocks. This effect can cause these tasks to stall until ASTs are reenabled.

Example 7–11 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;
```

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.

- Do not create or wait for dependent tasks. Execute any of the tasking operations described in Chapter 9 of the *DEC Ada Language Reference Manual* (for example, make entry calls, execute accept or select statements, and so on).
- 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$HIBER suspends execution of a OpenVMS process. The DEC Ada tasking implementation uses SYS$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$HIBER request. The SYS$WAKE you enter may be consumed by the tasking implementation call to SYS$HIBER, and your hibernating process will not wake up.

SYS$EXIT causes an unconditional program exit. In particular, it does not wait for dependent tasks to terminate normally. By using SYS$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 as shown in Example 7–12.

Example 7–12 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;
```

(continued on next page)

Example 7–12 (Cont.) Unpredictability of SYS$EXIT

```
   procedure DO_EXIT(STATUS      : out CONDITION;
                     EXIT_STATUS : in CONDITION := 1);
   pragma INTERFACE(OpenVMS, 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 OpenVMS exit handlers. 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 7.7.3 for the restrictions on using AST handlers.)

Specifically, an OpenVMS 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.

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.

7.6 Calling DECthreads Routines from Tasks (Alpha Systems Only)

Note

Although DECthreads is available on VAX systems, you cannot call DECthreads routines from Ada programs on VAX systems. Attempts to call DECthreads routines from VAX Ada programs do not work and are not supported.

On Alpha systems, calls to DECthreads routines are used to implement Ada tasking. You can use Ada tasks and foreign code that calls DECthreads routines together.

Because of the interaction between Ada tasks and DECthreads routines, direct calls to DECthreads routines are *not* supported in code running in the context of an Ada task. For example, the following interactions can cause problems:

- By calling routines such as CMA_ALERT_DEFER_ALL or PTHREAD_SETCANCEL (routines that affect alert delivery or the ability to cancel threads), you could affect the behavior of the abort statement. In fact, routines such as these could inadvertently turn synchronous abort into asynchronous abort (see Section 7.4.6).
- CMA_(UN)LOCK_GLOBAL and PTHREAD_(UN)LOCK_GLOBAL use the same mutex as the subprograms in the package SYNCHRONIZE_NONREENTRANT_ACCESS.
- CMA_STACK_LIMIT_CHECK_NP may give incorrect results when called from Ada.
- By calling CMA_THREAD_EXIT_ERROR, CMA_THREAD_EXIT_NORMAL, and PTHREAD_EXIT, you may prevent the execution of important cleanup operations, such as deleting collections, waiting for tasks, and so on. Furthermore, you could corrupt the Ada run-time library data structures.
- Routines that dynamically modify priority or scheduling policy such as CMA_THREAD_SET_PRIORITY, PTHREAD_SETPRIO, CMA_THREAD_SET_SCHED, and PTHREAD_SET_SCHEDULER invalidate the Ada run-time library's knowledge of task priority.

- PTHREAD_CLEANUP_PUSH and PTHREAD_CLEANUP_POP can cause problems for Ada exception handling. (These are actually not DECthreads routines but C macros. So, you cannot call them directly from an Ada program. However, your program could be calling some non-Ada code that does call these macros.)
- By specifying Ada subprograms or routines that call Ada subprograms as the value of the destructor parameter to the CMA_KEY_CREATE or PTHREAD_KEYCREATE routines, you may cause problems during task (or thread) termination.

For more information about DECthreads, see the *Guide to DECthreads*.

7.7 Handling Asynchronous System Traps (ASTs)

ASTs are a way for the OpenVMS operating system or OpenVMS RMS to notify a process (which may be actively executing instructions) that some event has occurred. Many OpenVMS system services let you 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 OpenVMS system service SYS$QIOW makes the process wait until the service completes, but the asynchronous form, SYS$QIO, does not. Both forms let you 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 OpenVMS 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 7.5.1.

7.7.1 The Pragma AST_ENTRY and the AST_ENTRY Attribute

You handle ASTs in DEC 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 *DEC 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 it 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 this priority or the priority of the accepting task, whichever is higher.

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);
 . . .
```

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 DEC Ada run-time library. 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 *DEC Ada Language Reference Manual*) and may not occur immediately. The rendezvous is performed *after* the special service routine has returned, so other ASTs are not inhibited. (This behavior is required by the nature of ASTs and the nature of Ada rendezvous.)

If the entry call is made directly from the AST service routine, no other ASTs can be delivered until the rendezvous is completed; unpredictable deadlocks may result. Such deadlocks may still develop if a non-Ada program were to call an Ada program from an AST service routine. See Section 7.7.3 for more information.

7.7.2 Constraints on Handling ASTs

Any AST is ignored if it is delivered to a task that is completed or abnormal. 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 DEC 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 DEC 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 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 C for more information on this package and its operations.

7.7.3 Calling Ada Subprograms from Non-Ada AST Service Routines

Be 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 (DEC 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 is 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 DEC 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 OpenVMS environment.

All of this occurs because the OpenVMS operating system *implicitly* disables all AST delivery while an AST-handling routine is active.

You should handle ASTs in DEC Ada as described in Section 7.7.1. 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).

  Your routine must not use any of the input-output operations defined in Chapter 14 of the *DEC Ada Language Reference Manual* or in the DEC Ada package TASKING_SERVICES.
- Execute a delay statement.
- Propagate an unhandled exception.
- Execute any of the tasking operations described in Chapter 9 of the *DEC Ada Language Reference Manual* (for example, make entry calls, execute accept or select statements, and so on).

  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.

7.7.4 Examples of Handling ASTs from Ada Programs

Examples Example 7–13 and Example 7–14 show the use of the pragma AST_ENTRY and the AST_ENTRY attribute.

Example 7–13 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;
```

(continued on next page)

Example 7–13 (Cont.) Simple Use of the Pragma AST_ENTRY and the AST_ENTRY Attribute

```
begin

   -- 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 7–14 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;

   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;
```

(continued on next page)

Example 7–14 (Cont.) Using an AST Entry to Intercept a Ctrl/C

```
   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;

   -- 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));
```

(continued on next page)

Example 7–14 (Cont.) Using an AST Entry to Intercept a Ctrl/C

```
   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;
```

7.8 Measuring and Tuning Tasking Performance

When you use tasks in your program, you must frequently trade off 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 time slicing is in effect (see Section 7.3), you are increasing responsiveness at the expense of more task-switching overhead. On VAX systems, smaller values of the time-slice interval represent higher amounts of this overhead.

Similarly, if you assign a higher priority to a task (see Section 7.3), you are opting for responsiveness rather than throughput. Assigning a higher priority to some task invariably means that the program performs 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, DEC Ada provides the debugger commands SHOW TASK/STATISTICS and SHOW TASK/FULL. DEC Ada also provides the package GET_TASK_INFO to let you obtain task information. See *Developing Ada Programs on OpenVMS Systems* for information on debugging Ada tasks and using the package GET_TASK_INFO.

8

Improving Run-Time Performance

To write DEC 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

Data alignment often affects the efficiency of the system you are working on. The DEC Ada pragma COMPONENT_ALIGNMENT lets you change the default alignment for components of array and record types. See Chapter 13 of the *DEC Ada Language Reference Manual* and Section 1.2 of this manual for more information about data representation and data optimization.

8.1 Compiler Optimizations

The DEC 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
- Strength reduction in loops
- 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
- Instruction scheduling (on Alpha systems)

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
- Evaluation of other compile-time constant expressions that may not be considered 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

8.2 Using the Pragma INLINE

To let you expand subprograms inline and decrease the amount of time spent in making subprogram calls, the Ada language provides the pragma INLINE. In DEC 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 8.2.1 gives the conditions for the explicit use of this pragma, and Section 8.2.2 gives the conditions under which implicit inline expansion takes place.

See Section 8.2.3 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
    6      function "+" (X,Y: INTEGER) return INTEGER renames STANDARD."+";
    7      VAR1,VAR2: T := 3;
    8
    9      function "+" (X,Y: T) return INTEGER is
...........1
%I, (1) Pragma INLINE at line 13 applies to function +
%I, (1) Code generation suppressed for function +, which is always
    expanded inline [LRM 6.3.2; RTR 8.2.1]

   10      begin
   11         return INTEGER(X*Y);
   12      end;
   13      pragma INLINE("+");
   14
   15   begin
   16      PUT(VAR1+VAR2);
.............1
%I, (1) Call of procedure PUT in INTEGER_TEXT_IO at line 2 (from TEXT_IO
    at line 148) expanded inline [LRM 6.3.2; RTR 8.2]
...............2
%I, (2) Call of function + at line 9 expanded inline [LRM 6.3.2; RTR 8.2]

   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 OpenVMS Systems* for more information on these commands and this qualifier.

Chapter 6 of the *DEC Ada Language Reference Manual* gives the syntax and placement rules for this pragma.

8.2.1 Explicit Use

Inline expansion of a subprogram occurs each time the subprogram is called. You can explicitly cause a subprogram to be expanded inline by specifying it in a pragma INLINE. Then, the call to the subprogram is expanded inline provided that the following are true:

- The subprogram 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.) The inline expansion of a subprogram body from a unit in the compilation library creates a dependence on that unit.

A subprogram declaration or body is inlinable under the following conditions:

- The parameters are of any type except the following:
  - A task type
  - A composite type that has components of a task type
- Function results are 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

When you use the pragma INLINE for an implicit operator declaration or for a derived subprogram declaration, it has the following effect:

- 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.
- 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, a generic subprogram declaration, or 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 do 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 8.2.2) 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 subunit is not 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 8.3.1 for more information.)

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, 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 8.2.2), then code is not generated.

8.2.2 Implicit Use

The DEC 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 8.2.1).
- 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 OpenVMS 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 OpenVMS Systems* for more information on how inline expansion affects unit dependences, obsolete units, and recompilation.

8.2.3 Pragma INLINE Examples

The following sections show some special cases that use the pragma INLINE and give examples of using it with generics. In particular, 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.

8.2.3.1 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;
```

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. 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;
```

The procedure INLINED_F is expanded inline in the call from the procedure USE_INLINED_F.

8.2.3.2 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 8.2.1). 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 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 8.2.1).

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 8.3.1.) The dependence means that if the body for the generic procedure GEN_PROCEDURE is later compiled again or replaced, the procedure USE_GEN_PROCEDURE becomes obsolete and needs to be recompiled. See *Developing Ada Programs on DEC OSF/1 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 8.2.1). 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;
```

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). The call to INLINE_INT.DECLARE_VAR expands DECLARE_ VAR inline, and the call to INLINE_ARR.DECLARE_VAR does not.

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.

8.3 Making Use of Generics

DEC Ada offers a number of features that let you 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 8–1 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 OpenVMS 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 8–1 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

8.3.1 Using the Pragma INLINE_GENERIC

DEC 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 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. For example:

```
procedure USE_INTEGER is new USE_ITEM(INTEGER);
pragma INLINE_GENERIC(USE_INTEGER);
```

Chapter 12 of the *DEC Ada Language Reference Manual* gives the syntax and rules for using the pragma INLINE_GENERIC.

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 OpenVMS 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 OpenVMS 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.

8.3.2 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. 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 *DEC Ada Language Reference Manual* gives the syntax and rules for using the pragma SHARE_GENERIC. 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 OpenVMS Systems*).

Generic code sharing provides the following 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. Similarly, to write an input-output subsystem where files of many different types needed to be supported, C or assembly language 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. Generic code sharing eliminates one of the possible advantages of a less strongly typed language over Ada.
- It lets Ada programmers 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);

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 occurs only if the code that is generated for one instance is similar to the code generated for another. The execution times for shared code

are often similar to those for nonshared code, particularly for larger generic packages.

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. The size of your program increases 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.

8.3.3 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 DEC 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 are done only once (when the package is initially compiled and added to your program library), not each time you use them.

DEC 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 2 and Appendix C 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.

8.4 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 DEC Ada program on a OpenVMS system.

To decrease the program's CPU time on a particular processor, you can make the following 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 the following 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 DEC Performance and Coverage Analyzer (PCA) is an optional layered product, which is also included among the DECset tools. It measures the performance characteristics of user-mode programs running on OpenVMS systems. While the following discussions may offer some general assistance, the techniques they propose are best used in conjunction with DEC PCA.

To use DEC PCA, you must have it installed on your system, and you must compile and link your program with /DEBUG qualifiers in effect. See the DEC PCA documentation for information on how to use DEC PCA.

8.4.1 Decreasing the CPU Time of a DEC Ada Program

The first step in decreasing the CPU time of a DEC Ada program is to use DEC 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 do not gain much of a performance improvement by suppressing checks, explicitly expanding subprograms inline, or otherwise writing anything other than straightforward Ada code. 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, DEC 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 debugger or by examining a listing file that you have produced with the /LIST/MACHINE_CODE qualifier at compile time.

8.4.1.1 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 then performs the check outside of the loop, and uses 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 *DEC 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.

8.4.1.2 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 OpenVMS 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 small subprograms in other units, the cost of the run-time check to see if the subprogram body has been elaborated may be significant. When the compiler does not otherwise optimize away access-before-elaboration checks, 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 PKG;

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

-- Second compilation unit.
--
package body PKG is

   I : INTEGER := 0;
```

```
   function TRIVIAL return INTEGER is
   begin
      I := I+1;
      return I;
   end;

end PKG;

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

-- 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;       -- Note 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;
```

8.4.1.3 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, consider the following assignment:

```
A := B & (1 .. A'LENGTH - B'LENGTH => ' ');
```

You can replace this assignment with the following operations:

```
A(1 .. B'LENGTH) := B;
for I in INTEGER'(B'LENGTH + 1) .. A'LENGTH loop
   A(I) := ' ';
end loop;
```

- Rather than using aggregates, especially those involving run-time expressions, build values in place. For example, consider this single operation:

  ```
  A := (I*I, 2*J, K+0.3);
  ```

  You can replace this operation with the following 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 it with code like the following:

  ```
  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, you should replace the following code:

  ```
  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;
```

Here is a possible replacement:

```
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 let the compiler 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;
```

8.4.2 Decreasing the Elapsed Time of a DEC 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 8.4.1. 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.

When using different or more CPUs:

- If you obtain a faster CPU, your program's run-time performance improves 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 DEC Ada program into multiple DEC Ada programs and then using either networking or shared global sections to communicate the data between them.

  Chapter 9 includes a section with an example program that shares memory between one or more CPUs on a OpenVMS system.

8.4.2.1 Controlling Paging Behavior

Experience has shown that, in general, the OpenVMS 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 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. This effect occurs in an opposite way in FORTRAN where it is desirable to change the first index the fastest.

8.4.2.2 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 *OpenVMS Record Management Services Reference Manual* for more information on RMS routines. See the *VMS System Services Volume* for more information on OpenVMS system service routines.

8.4.2.3 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 accesses the CPU of the computing task, gets its next input-output started, and then waits—returning control to the computing task. If the computing task is given the higher (or even the same) priority, the input-output-bound task is not able to start its input-output as soon as possible, and its elapsed time is 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
   P: ACCESS_PICTURE;
begin
   loop
      -- {compute a new picture};
      GRAPHICS_ENGINE.PUT_PICTURE(P);
   end loop;
end;
```

Most of the computation in this example is 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 is driven at full speed.

9

Additional Programming Considerations

This chapter documents DEC Ada programming considerations that may not be immediately obvious but that may affect the run-time behavior or performance of your DEC Ada programs. It also documents the use of some of the low-level, system-specific features of DEC Ada.

9.1 Working with Address Values

To let you 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 DEC 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, 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 DEC Ada compiler implicitly treats 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 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. Because X has been marked as 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;
```

9.2 Unchecked Conversion of Access Types

DEC Ada uses a virtual address to represent the value of an access type. However, this address is not necessarily the address of the accessed object. In particular:

- In the case of an access type whose subtype is an unconstrained array type, the address may be that of a descriptor of the accessed value. Unchecked conversion between an access type and an unconstrained array type does not work.
- In some cases, the size of the accessed object is not the same as the size of a similar object that would otherwise be allocated statically or on the stack. For example, access to noncomposite types without size representation clauses usually results in the accessed value occupying the correct amount of storage. However, if a noncomposite type has a size representation clause, then the size of the accessed value may be wrong.

Therefore, unchecked conversion between access types and the type SYSTEM.ADDRESS is guaranteed to work in DEC Ada only when the access type's subtype indication is either a record type (which may or may not have a constraint imposed on it) or a constrained array subtype.

A common use of this kind of unchecked conversion is to translate private types to or from arrays of integers for input-output purposes. The recommended method for performing this primitive operation is as follows:

```
generic
   type T is private;
package TRANSLATE_PRIVATE is
 . . .
end TRANSLATE_PRIVATE;

with SYSTEM, UNCHECKED_CONVERSION;
package body TRANSLATE_PRIVATE is
   type R is record
      C : T;
   end record;
   type A is access R;
   function TO_A is
      new UNCHECKED_CONVERSION(SYSTEM.ADDRESS, A);
    . . .
   BUFFER : array (1 .. 100) of INTEGER;
   CONVERTED_VALUE : T;
begin
    . . .
   CONVERTED_VALUE := TO_A(BUFFER'ADDRESS).all.C;
    . . .
end TRANSLATE_PRIVATE;
```

There are two important consequences of using a private type (in this example, T) as the subtype indication of a record component. First, any private type can be taken and fitted into a supported case. Second, an unconstrained array type cannot be used to instantiate the generic. (It is illegal.) This second case causes the most trouble because the value of the access type is not represented by the address of the accessed object.

You can avoid the approach of the preceding example and obtain a better program structure. You can pass subprograms to the generic that are specifically designed to convert the formal type to or from the type that is to be input or output. The result is fully portable code that uses no implementation-specific features, and the package exporting the type has full control over the external representation of the type. Example 9–1 is an example of this technique.

Example 9–1 A Portable Technique for Reading and Writing Private Types

```
-- G_IO is a generic input-output package.  It is
-- used for writing values of a variety of types
-- to a single file, and then reading them back.
--
generic
   -- The type for which input-output is to be
   -- provided.
   --
   type T is private;

   -- The functions to convert values of type T to and
   -- from their external representations as values of
   -- the type STRING.
   --
   with function TO_T(S : STRING) return T is <>;
   with function TO_STRING(ITEM : T)
      return STRING is <>;

package G_IO is
   procedure PUT(ITEM : in  T);
   procedure GET(ITEM : out T);
end G_IO;

package body G_IO is
   procedure PUT(ITEM : in  T) is
   begin
       . . .
   end;

   procedure GET(ITEM : out T) is
   begin
       . . .
   end;
end G_IO;
```

(continued on next page)

Example 9–1 (Cont.) A Portable Technique for Reading and Writing Private Types

```
-- PRIV_EXPORTER declares the private type T, whose
-- internal representation is unknown to the outside
-- world.  To make it possible to do input-output
-- operations on the type T, PRIV_EXPORTER provides
-- two subprograms, TO_T and TO_STRING, to convert
-- values of the type T to values of the type STRING.
-- The values of the type STRING can then be read
-- from and written to a file.  The exact contents
-- of these strings is deliberately not defined.
--
package PRIV_EXPORTER is
   type T is private;
   function TO_T(S : STRING) return T;
   function TO_STRING(ITEM : T) return STRING;
   procedure INIT(X : in out T);
private
   type T is new INTEGER;
end PRIV_EXPORTER;

package body PRIV_EXPORTER is
   function TO_T(S : STRING) return T is
   begin
      return T'VALUE(S);
   end TO_T;

   function TO_STRING(ITEM : T) return STRING is
   begin
      return T'IMAGE(ITEM);
   end TO_STRING;

   procedure INIT(X : in out T) is
   begin
      X := 0;
   end INIT;
end PRIV_EXPORTER;
```

(continued on next page)

Example 9–1 (Cont.) A Portable Technique for Reading and Writing Private Types

```
-- The procedure EG shows the input and output of
-- values of the type PRIV_EXPORTER.T.
--
with PRIV_EXPORTER, G_IO;
use  PRIV_EXPORTER;
procedure EG is
   X : PRIV_EXPORTER.T;
   package T_IO is new G_IO(PRIV_EXPORTER.T);
begin
   INIT(X);
   T_IO.PUT(X);
   T_IO.GET(X);
end EG;
```

9.3 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.

9.3.1 The VAX Device and Processor Register and Interlocked Operations (VAX Systems Only)

Applications accessing OpenVMS input-output space or using shared memory have special restrictions on which VAX instructions can be used. You can force the DEC Ada compiler to meet these restrictions by using the operations defined in the package SYSTEM. See Table 9–1.

Table 9–1 VAX Instructions Provided in the Predefined 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)

(continued on next page)

Table 9–1 (Cont.) VAX Instructions Provided in the Predefined Package SYSTEM

Operation	Equivalent VAX Instruction
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 *DEC 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 9–2 shows one method of implementing a queue using the interlocked queue operations. The interlocked queue operations all require quadword alignment of the queue elements. To satisfy this requirement, you can use Ada alignment clauses. However, DEC Ada allows some 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 *DEC Ada Language Reference Manual* for more information on the alignment clauses and the allowed restrictions. See Chapter 1 for additional information on the use of alignment clauses and for information on how and where storage for objects is allocated.

Example 9–2 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 9–2 (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 9–2 (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;
```

9.3.2 Unsigned Types in the Package SYSTEM

Unsigned types declared in the package SYSTEM 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 ($-2^{31}..2^{31} - 1$) –16#80000000#..16#7FFFFFFF#

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:

- OpenVMS 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 *DEC 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 $-2^{32} - 1$).

When the hardware does not support all arithmetic operations on quadwords, this implementation is inefficient. So the DEC Ada implementation defines UNSIGNED_LONGWORD as if it were an integer type with the range $-2^{31}..2^{31} - 1$ 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. Therefore, 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, the following declaration raises 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. Therefore, 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 do 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;
```

9.4 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, 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 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. DEC Ada, in fact, does just this. 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 DEC Ada, must not exceed the implementation-defined limit for type INTEGER ($2^{31} - 1$ Otherwise, CONSTRAINT_ERROR must be raised. (See Chapter 11 and Appendix F of the *DEC 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. So, the run-time computation of the maximum size of X ($2^{31} - 1 + 4 + 4$) raises CONSTRAINT_ERROR. Replacing NATURAL'LAST with NATURAL'LAST – 8 in the definition of STRING_SIZE allows the

maximum size to be computed. The compiler then attempts 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 is 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 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!");
```

9.5 Assigning Array Values

When you assign array values, consider the specific rules about assignments listed in Chapter 5 of the *DEC Ada Language Reference Manual*. In particular, 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 *DEC Ada Language Reference Manual*):

```
S2 := S1(6..10);
```

The expression S1(6..10) is implicitly converted to the subtype of the left side (STRING(1..5)). 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 *DEC 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 *DEC 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 *DEC 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. 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. So, 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, 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;
```

9.6 Sharing Memory Between CPUs

Example 9–3 shows how to write a DEC Ada program that shares memory between two or more DEC Ada programs running on one or more CPUs on a OpenVMS system.

The program uses OpenVMS 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 OpenVMS lock manager may significantly improve performance.

Example 9–3 Sharing Memory Between Two or More Programs Running on One or More 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.
--
--  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, STARLET;
package SHARED_MEMORY_DATA_TYPES is

   type SHARED_OBJECTS_TYPE is
      record
         VALUE_AVAILABLE : BOOLEAN;
         VALUE           : INTEGER;
      end record;
```

(continued on next page)

Example 9–3 (Cont.) Sharing Memory Between Two or More Programs Running on One or More CPUs

```
   for SHARED_OBJECTS_TYPE use
      record
         VALUE_AVAILABLE at 0 range 0 .. 7;
         VALUE           at 4 range 0 .. 31;
      end record;

end SHARED_MEMORY_DATA_TYPES;

--  Next, write a procedure that can call the VMS system service
--  SYS$CRMPSC to create memory that is shared between two processes.
--  In this example, a groupwide section is either created (if it
--  did not already exist) or is mapped. The caller is returned two
--  pieces of information:
--
--     o  The address of the section (in this process's address
--        space; it may be at a different address in a different
--        process)
--
--     o  A boolean value indicating whether or not this was the
--        creating call to SYS$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 9–3 (Cont.) Sharing Memory Between Two or More Programs Running on One or More 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 9–3 (Cont.) Sharing Memory Between Two or More Programs Running on One or More 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 9–3 (Cont.) Sharing Memory Between Two or More Programs Running on One or More 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 the system service routine
   -- SYS$ENQW (enqueue lock request and wait) causes all
   -- memory modifications to be flushed through the multiCPU caches
   -- and become visible to all participating processes.
   --
   -- The use of the TASKING_SERVICES version of this routine
   -- (TASKING_SERVICES.TASK_ENQW) 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;

   FAILED_LOCK_REQUEST : exception;

end SHARED_MEMORY;

with CONDITION_HANDLING, SYSTEM, STARLET, TASKING_SERVICES;
use STARLET, TASKING_SERVICES;
package body SHARED_MEMORY is

   STATUS            : CONDITION_HANDLING.COND_VALUE_TYPE;
   LOCK_STATUS_BLOCK : LOCK_STATUS_BLOCK_TYPE;

   procedure ACQUIRE is
   begin
      TASK_ENQW (STATUS => STATUS,
                 LKMODE => LCK_K_EXMODE,
                 LKSB   => LOCK_STATUS_BLOCK,
                 FLAGS  => LCK_M_CONVERT);
```

(continued on next page)

Example 9–3 (Cont.) Sharing Memory Between Two or More Programs Running on One or More CPUs

```
      if (not CONDITION_HANDLING.SUCCESS(STATUS)) or
         (not CONDITION_HANDLING.SUCCESS(LOCK_STATUS_BLOCK.STATUS))
      then
         raise FAILED_LOCK_REQUEST;
      end if;
   end;

   procedure RELEASE is
   begin
      TASK_ENQW (STATUS => STATUS,
                 LKMODE => LCK_K_NLMODE,
                 LKSB   => LOCK_STATUS_BLOCK,
                 FLAGS  => LCK_M_CONVERT);
      if (not CONDITION_HANDLING.SUCCESS(STATUS)) or
         (not CONDITION_HANDLING.SUCCESS(LOCK_STATUS_BLOCK.STATUS))
      then
         raise FAILED_LOCK_REQUEST;
      end if;
   end;

begin
   TASK_ENQW (STATUS => STATUS,
              LKMODE => STARLET.LCK_K_NLMODE,
              LKSB   => LOCK_STATUS_BLOCK,
              RESNAM => "INTERLOCK");
   if (not CONDITION_HANDLING.SUCCESS(STATUS)) or
      (not CONDITION_HANDLING.SUCCESS(LOCK_STATUS_BLOCK.STATUS))
   then
      raise FAILED_LOCK_REQUEST;
   end if;

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;
```

(continued on next page)

Example 9–3 (Cont.) Sharing Memory Between Two or More Programs Running on One or More CPUs

```
         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;

-- 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;
```

A

DEC Ada Predefined Instantiations

For convenience, and for the purpose of saving compilation time and object code space, DEC Ada predefines the instantiations of some commonly used generic packages. See Table A–1.

Table A–1 Predefined Instantiations of 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[1]	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[1]	MATH_LIB	LONG_LONG_FLOAT

[1]On VAX systems only.

The representation used for the type LONG_FLOAT in these packages is G_floating. 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:

- For the G_floating representation to be available, the value of the pragma FLOAT_REPRESENTATION must be VAX_FLOAT. This value is the default on Alpha systems. It is the only value on VAX systems.

On Alpha systems you can use either the pragma FLOAT_REPRESENTATION or the ACS SET PRAGMA command to set this value.

- G_floating is the default whenever you create or reinitialize a program library or sublibrary and the value of the pragma FLOAT_REPRESENTATION is VAX_FLOAT.
- You can set the representation for LONG_FLOAT either with the DEC 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 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 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 the *DEC Ada Language Reference Manual* for more information on the pragma LONG_FLOAT. See Section 1.1.3.1 of this manual for more information on the pragma FLOAT_REPRESENTATION. See *Developing Ada Programs on OpenVMS Systems* for more information on ACS commands, compiling Ada programs, and the implied recompilation of the predefined STANDARD environment.

See Chapter 2 of this manual for information on using the predefined instantiations for TEXT_IO.INTEGER_IO and TEXT_IO.FLOAT_IO. See Chapter 5 of this manual for information on using the predefined instantiations of MATH_LIB.

B

Implementation Details Related to Mixed-Language Programs on OpenVMS Systems

When writing mixed-language programs, you may need more detailed information about how parameter-passing mechanisms are implemented. This appendix contains information on constrainedness bits, the area control block, and descriptors. See Chapter 4 for general information on mixed-language programming.

B.1 Constrainedness Bits

For exported subprograms, if a parameter is of a record type that 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 value as an extra 32-bit quantity at the end of 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. For example, in the following fragment, the parameter X is the first parameter that requires a constrainedness bit; parameter Z is the second.

```
type T is (ONE, TWO, THREE);

type R(D: T:=ONE) is
   record
      case D is
         when ONE => null;
         when others => E: INTEGER := 10;
      end case;
   end record;
```

```
procedure P(I: INTEGER;
                   X: in out R;
                   Y: FLOAT;
                   Z: out R);
```

If this procedure is exported and called, the calling routine will need to pass an extra 32-bit integer or 32-element boolean array parameter that sets bits 0 and 1.

Example B–1 shows how to code a call to an Ada subprogram that requires constrainedness bits. The code for importing the Ada subprogram is written in Ada, but could be written in another language.

Example B–1 Calling an Ada Subprogram and Passing Constrainedness Bits

```
package ADA_EXPORT is

   type R1(D : BOOLEAN) is
      record
         null;
      end record;

   for R1 use
      record
         D at 0 range 0 .. 7;
      end record;

   for R1'SIZE use 8;

   procedure EXPORT_COPY(FROM1, FROM2 : in  R1;
                         TO1,   TO2   : out R1);
   pragma EXPORT_PROCEDURE(INTERNAL  => EXPORT_COPY,
                           EXTERNAL  =>"Copy",
                           MECHANISM =>(REFERENCE, REFERENCE,
                                        REFERENCE, REFERENCE));

end ADA_EXPORT;

package body ADA_EXPORT is

   procedure EXPORT_COPY(FROM1, FROM2 : in  R1;
                         TO1,   TO2   : out R1) is
   begin
      TO1 := FROM1;   -- Assignment raises CONSTRAINT_ERROR unless
      TO2 := FROM2;   -- TO is unconstrained or TO.D = FROM.D.
   end;

end ADA_EXPORT;
-------------------------------------------------------------
```

(continued on next page)

Example B–1 (Cont.) Calling an Ada Subprogram and Passing Constrainedness Bits

```
package SOME_IMPORT is
   type R2 is
      record
         D : BOOLEAN;
      end record;

   for R2 use
      record
         D at 0 range 0..7;
      end record;

   for R2'SIZE use 8;

   type BOOLEAN_VECTOR_32 is array(1 .. 32) of BOOLEAN;
   pragma PACK(BOOLEAN_VECTOR_32);

   procedure IMPORT_COPY(FROM1, FROM2 : in R2;
                         TO1,   TO2   : in out R2; -- 'in out' because
                                                   -- TO1.D/TO2.D are read.
           IS_CONSTRAINED : BOOLEAN_VECTOR_32);

   pragma INTERFACE(ADA, IMPORT_COPY);
   pragma IMPORT_PROCEDURE(INTERNAL  => Import_Copy,
                           EXTERNAL  =>"Copy",
                           MECHANISM =>(REFERENCE, REFERENCE,
                                        REFERENCE, REFERENCE,
                                 IS_CONSTRAINED => VALUE));

end SOME_IMPORT;

with SOME_IMPORT; use SOME_IMPORT;
with ADA_EXPORT; use ADA_EXPORT;
procedure CALL_IMPORT is

   type E is (FROM1, FROM2, TO1, TO2, CONSTRAINED1, CONSTRAINED2, DO_IT);

   B : array(FROM1 .. CONSTRAINED2) of BOOLEAN;

   ITEMS : array(FROM1 .. TO2) of R2;
```

(continued on next page)

Example B–1 (Cont.) Calling an Ada Subprogram and Passing Constrainedness Bits

```
begin

   IMPORT_COPY(ITEMS(FROM1), ITEMS(FROM2),
               ITEMS(TO1), ITEMS(TO2),
               BOOLEAN_VECTOR_32'(B(CONSTRAINED1),
                                  B(CONSTRAINED2),
                                  others => FALSE));

end CALL_IMPORT;
```

B.2 Area Control Block

When exporting a DEC Ada function, note that if the function result is of an unconstrained array type (including unconstrained string types), or if the result is of a large unconstrained record type with discriminants, 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 B–1.

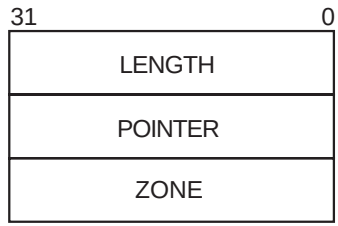
In some cases, the area control block may be followed by a descriptor. In such a case, the calling routine must initialize the area control block. However, the calling routine does not need to initialize the descriptor (if a descriptor is involved) or allocate storage for the result. The called function 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 to return the result. The caller is responsible for releasing the storage for that result after the storage has been used.

B.3 Descriptors

Descriptors describe certain kinds of data uniformly. In DEC Ada, a descriptor consists of a set of contiguous fields that, in general, contain the following information:

- The length of the data item
- The data type represented by the descriptor

Figure B–1 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 (if ZONE is −1). (The called routine has the option of doing either if the previously allocated storage is sufficient.)

3. If ZONE is zero, then dynamic memory is used.

4. 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.

5. 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

- An integer value (class code), which identifies the format and interpretation of other fields in the descriptor
- The address of the first byte of the data element described

The descriptors supported by DEC Ada are a subset of the descriptors defined by the OpenVMS calling standard (see *OpenVMS Calling Standard*).

Note

If you choose to specify class names when you use the DESCRIPTOR mechanism in an import or export pragma, you must observe the type requirements described in Sections Section B.3.1 to Section B.3.7.

You can use any of the descriptor classes listed in Table B–1 for passing parameters of different data types.

Note

This manual uses the following terms:

- 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 *string* is any one-dimensional array of a discrete type whose components occupy successive, unsigned bytes.

Table B–1 Descriptor Classes Allowed for Passing Ada Parameters

Descriptor Class	Definition
UBS	Unaligned bit string
UBSB	Unaligned bit string with arbitrary bounds
UBA	Unaligned bit array
S	String; also imported scalar or access type parameter; not allowed for exported scalar or access type parameter

(continued on next page)

Table B–1 (Cont.) Descriptor Classes Allowed for Passing Ada Parameters

Descriptor Class	Definition
SB	String with arbitrary bounds
A	Contiguous array
NCA[1]	Noncontiguous array

[1]Not allowed for parameters of imported routines that are identified by the language ADA in the pragma INTERFACE or for parameters of exported subprograms.

When descriptors are used to pass parameters or return function results in an Ada program, the DEC Ada compiler generates the descriptor and supplies the necessary information. For certain array parameters, the DEC Ada compiler automatically chooses one of the descriptors in Table B–1.

If you use the DESCRIPTOR mechanism name in an imported or exported subprogram specification, but omit the class name, the DEC Ada compiler chooses an appropriate class depending on the Ada parameter type.

The following sections discuss the descriptors supported by DEC Ada. See the *OpenVMS Calling Standard* for complete information on these descriptors, as well as for information on the larger set of OpenVMS descriptors.

B.3.1 UBS Descriptor

The UBS descriptor describes unaligned bit-string data.

When you specify the UBS descriptor class for an imported or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- The base type must be a bit string.
- For imported routines that are identified by the language ADA and for exported subprograms, the base type must be a bit string whose lower bound is equal to 1.
- A run-time descriptor check may occur to ensure that the actual array parameter has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised.

B.3.2 UBSB Descriptor

The UBSB descriptor describes unaligned bit-string data, where the string is viewed as a one-dimensional bit array with user-specified bounds.

When you specify the UBSB descriptor class for an imported or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- The base type must be a bit string.
- A run-time descriptor check may occur to ensure that the actual parameter or result has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised.

B.3.3 UBA Descriptor

The UBA descriptor describes an array of unaligned bit strings.

When you specify the UBA descriptor class for an imported or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- The base type must be an array.
- For imported routines that are identified by the language ADA and for exported subprograms, the parameter or function result must be a bit string or a bit array.
- A run-time descriptor check may occur to ensure that the size of each component of the actual parameter or result 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 or result array components are unaligned (the formal parameter or result type has been declared with pragma PACK). If the array components are byte aligned, use descriptor class A.

B.3.4 S Descriptor

The S descriptor describes scalar data, access types, address types, and fixed-length strings.

When you specify the S descriptor class for an imported or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- For an imported subprogram parameter or function result, the base type must be a scalar, access, or address type, or it must be a one-dimensional

array of 8-bit unsigned components (for example, a string type or an array of packed 8-bit components).

- For an exported subprogram parameter or function result, the base type must be a one-dimensional array of 8-bit unsigned components (for example, a string type or an array of packed 8-bit components).
- For an imported routine that is identified by the language ADA or for an exported subprogram, if the base type is an array, the lower bound must be equal to 1.
- A run-time descriptor check may occur to ensure that the actual array parameter or result has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised.

B.3.5 SB Descriptor

The SB descriptor describes a fixed-length string, where the string is a one-dimensional array with user-specified bounds.

When you specify the SB descriptor class for an imported or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- For a routine that is not identified by the language ADA in the pragma INTERFACE, the base type must be a one-dimensional array of unsigned 8-bit components (a DEC Ada string type).
- For a routine that is identified by the language ADA or for an exported subprogram, the base type *must* be a one-dimensional array of 8-bit unsigned components.
- A run-time descriptor check may occur to ensure that the actual array parameter has no more than 65,535 components. If this check fails, then CONSTRAINT_ERROR is raised.

B.3.6 A Descriptor

The A descriptor describes contiguous arrays of atomic data types or contiguous arrays of fixed-length strings.

When you specify the A descriptor class for an imported routine or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- The base type can be any array type except a bit string or bit array type.

- A run-time descriptor check may be performed to ensure that the actual array parameter or result is byte aligned. If this check fails, then CONSTRAINT_ERROR is raised. In all other cases, a run-time descriptor check may be performed to ensure that the component size does not exceed 65,535 bytes. If this check fails, then CONSTRAINT_ERROR is raised.

For a one-dimensional array of unsigned 8-bit components that is not a string type, 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. Class A can be used where it is not known at compile time that there are always fewer than 65,535 components for all possible values of the type.

B.3.7 NCA Descriptor

The NCA descriptor describes an array where the storage of the array elements may be allocated with a fixed, nonzero number of bytes separating logically adjacent elements. The array may be noncontiguous. When you specify the NCA descriptor class for an imported or exported subprogram parameter or function result, the DEC Ada formal parameter or result must have the following type characteristics:

- For imported routines that are not identified by the language ADA in the pragma INTERFACE, the base type can be any array type except a bit string or bit array type.
- For imported routines that are identified by the language ADA or for exported subprograms, the NCA descriptor class is not allowed. In other words, because DEC 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.

B.3.8 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$K_CLASS_SB, the Ada compiler ensures that parameters passed by descriptor class DSC$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 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 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 DEC Ada). Because the Ada function is exported, it also accepts the string STR if the string is passed by DSC$K_CLASS_S descriptor. The %REF mechanism specifier in the Fortran routine guarantees that B isbe passed by reference.

C
DEC Ada Packages

DEC Ada provides the packages listed in Table C–1. As noted in Table C–1, this appendix and the *DEC 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/SPECIFICATION_ONLY 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 OpenVMS Systems* or type HELP ACS EXTRACT SOURCE at the OpenVMS prompt for more information.

Table C–1 DEC Ada Predefined Packages

Package Name[1]	Description
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.
ASSERT_EXCEPTIONS	Declares all exceptions that can be raised by instantiations of the package ASSERT_GENERIC.
ASSERT_GENERIC	Provides types and operations that allow you to insert and enable code-checking assertions in your Ada source code.

[1]Some package specifications appear in the *DEC Ada Language Reference Manual* and this appendix. All package specifications are available from the DEC Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table C–1 (Cont.) DEC Ada Predefined Packages

Package Name[1]	Description
AUX_IO_EXCEPTIONS	Defines the exceptions needed by the DEC Ada relative and indexed input-output packages.
CALENDAR	Provides time-related types and operations.
CDD_TYPES	Provides Ada equivalents for CDD/Repository data types; additional equivalents are in the packages STANDARD and SYSTEM.
CLI	Provides types and operations for calling OpenVMS Command Language Utility routines.
CONDITION_HANDLING	Provides types and operations needed to evaluate the condition values returned by system routines.
	Depends on the package SYSTEM.
CONTROL_C_INTERCEPTION	Establishes A DEC Ada Ctrl/C handler when it is elaborated.
C_TYPES	Collection of Ada type definitions and conversion operations that correspond to familiar types defined by the C language.
DIRECT_IO	Provides types and operations for working with direct files of uniform-type elements.
DIRECT_MIXED_IO	Provides types and operations for working with direct files of mixed-type elements.
DTK	Provides types and operations for calling the OpenVMS Run-Time Library DTK$ routines.
GET_TASK_INFO	Provides information about specific Ada tasks.
INDEXED_IO	Provides types and operations for working with indexed files of uniform-type elements.
INDEXED_MIXED_IO	Provides types and operations for working with indexed files of mixed-type elements.
IO_EXCEPTIONS	Defines exceptions needed by all of the input-output packages.
LBR	Provides types and operations for calling the OpenVMS Librarian Utility routines.
LIB	Provides types and operations for calling the OpenVMS Run-Time Library LIB$ routines.

[1]Some package specifications appear in the *DEC Ada Language Reference Manual* and this appendix. All package specifications are available from the DEC Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table C–1 (Cont.) DEC Ada Predefined Packages

Package Name[1]	Description
MATH_LIB	Provides a set of operations and exceptions that correspond to some of the OpenVMS Run-Time Library Mathematical Library routines and conditions.
MTH	Provides types and operations for calling the OpenVMS Run-Time Library MTH$ routines.
NCS	Provides types and operations for calling the National Character Set Utility routines.
OTS	Provides types and operations for calling the OpenVMS Run-Time Library OTS$ routines.
PPL	Provides types and operations for calling the OpenVMS Run-Time Library PPL$ routines.
RELATIVE_IO	Provides types and operations for working with relative files of uniform-type elements.
RELATIVE_MIXED_IO	Provides types and operations for working with relative files of mixed-type elements.
RMS_ASYNCH_OPERATIONS	Provides supporting operations for the package TASKING_SERVICES.
SEQUENTIAL_IO	Provides types and operations for working with sequential files of uniform-type elements.
SEQUENTIAL_MIXED_IO	Provides types and operations for working with sequential files of mixed-type elements.
SMG	Provides types and operations for calling the OpenVMS Run-Time Library SMG$ routines.
SOR	Provides types and operations for calling the Sort/Merge Utility routines.
STANDARD	Provides all the predefined types, operations, and exceptions defined by the language, as well as the additional DEC Ada types SHORT_INTEGER, SHORT_SHORT_INTEGER, LONG_INTEGER, LONG_FLOAT, and LONG_LONG_FLOAT.
STARLET	Provides the types, operations, constants, and so on that you need to call OpenVMS system service and RMS routines.

[1]Some package specifications appear in the *DEC Ada Language Reference Manual* and this appendix. All package specifications are available from the DEC Ada library of predefined units, ADA$PREDEFINED.

(continued on next page)

Table C–1 (Cont.) DEC Ada Predefined Packages

Package Name[1]	Description
STR	Provides types and operations for calling the OpenVMS Run-Time Library STR$ routines.
SYNCHRONIZE_NONREENTRANT_ ACCESS	Provides procedures that let you set up a locking protocol that prevent problems when you are calling routines that are not fully reentrant.
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 DEC Ada.
SYSTEM_RUNTIME_TUNING	Provides operations for changing system parameters that are normally controlled by the DEC Ada run-time library.
TASKING_SERVICES	Provides task-synchronous, process-asynchronous forms of some of the OpenVMS system services.
TEXT_IO	Provides types and operations for working with text files.
VAXELN_SERVICES	Interfaces for VAXELN routines; useful only with VAXELN Ada programs.

[1]Some package specifications appear in the *DEC Ada Language Reference Manual* and this appendix. All package specifications are available from the DEC Ada library of predefined units, ADA$PREDEFINED.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

X